12-2012

Parallel For Loops on Heterogeneous Resources

Frederick Edward Weber
University of Tennessee - Knoxville, fweber1@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Frederick Edward Weber entitled "Parallel For Loops on Heterogeneous Resources." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Gregory D. Peterson, Major Professor

We have read this dissertation and recommend its acceptance:

Robert J. Harrison, Micah Beck, Robert Hettich

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Parallel For Loops on Heterogeneous Resources

A Dissertation

Presented for the

The Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Frederick Edward Weber

December 2012

I dedicate this dissertation to my family and the faculty and staff who have supported me on every step of this journey.

Acknowledgements

I would like to thank Greg Peterson for being a fantastic advisor, my comittee members, and SCALE-IT for funding much of my graduate career.

Ça plane pour moi.

Abstract

In recent years, Graphics Processing Units (GPUs) have piqued the interest of researchers in scientific computing. Their immense floating point throughput and massive parallelism make them ideal for not just graphical applications, but many general algorithms as well. Load balancing applications and taking advantage of all computational resources in a machine is a difficult challenge, especially when the resources are heterogeneous. This dissertation presents the clUtil library, which vastly simplifies developing OpenCL applications for heterogeneous systems. The core focus of this dissertation lies in clUtil's ParallelFor construct and our novel PINA scheduler which can efficiently load balance work onto multiple GPUs and CPUs simultaneously.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 A Brief History of Heterogeneous Computing

The ideas and motivations behind heterogeneous computing predate history, harking back to the birth of civilization itself. Specialization of labor allows individuals to perform tasks and duties best suited for them while leaving tasks they perform poorly to others. Several millennia later, this theme is again emerging in processor design and computer architecture.

Heterogeneous computing, using multiple types of computing elements to solve problems, has existed for at least two decades now [1][2]. However, the idea of using multiple devices in tandem to accelerate applications wasn't even entirely novel then; coprocessors such as the Intel 8087 floating point unit, sound cards, and I/O cards existed before then.

The motivation for using multiple devices stems from the fact that each device can be application specific, allowing the collection of accelerators to each speed up a piece of the application. This allows for synergistic speedups in execution times, providing a cost effective and power efficient way to solve problems. Unfortunately, heterogeneous computing has several challenges including mapping the problem to the architectures involved, scheduling, and synchronization among others [3].

With the attack of the killer micros in the 1990s, supercomputing became dominated with cheap commodity processors [4]. Homogeneous computers such as the Intel Paragon [5], a large toroidal mesh of Intel i860 processors, came to dominate scientific computing. Other workstation processors including the DEC Alpha and SPARC also became common in large supercomputers [6][7]. In a separate vein, reconfigurable computing grew in interest in the 90s with the advent of the CHS2x4, the first reprogrammable hardware accelerator [8].

While not a commercial success, the CHS2x4 opened the gateway for a new era of heterogeneous computing. Companies such as SRC, DRC, Nallatech, and Cray began offering systems with FPGAs either in-socket or as an expansion card [9]. Reconfigurable hardware in tandem with a CPU allows developers to accelerate applications with custom circuits, potentially providing immense speedups over a CPU alone. Some [10] hailed reconfigurable computing as the end of the Von Neumann architecture, but this prophesy has yet to pass: instruction based computers still dominate in many areas of computer science. Developing hardware is a time consuming process that requires specialized specialized skillsets. Furthermore, clock rates and transistor efficiency are much lower on current FPGAs than custom ASICs [11]. These reasons allowed processors to remain the dominant force in scientific computing.

Processors have changed dramatically since the 90s. Earlier processor improvements focused mostly on Instruction Level Parallelism (ILP) and higher clock-rates. This had the effect of automatically improving legacy code's speed, since upgrading to a new processor yielded increased instruction throughput. This allowed software designers to reap the benefits of new architectures with no code changes, and usually even without recompilation. However, the silicon costs for additional instruction parallelism yielded diminishing returns [12] and the power wall [13] combined with an ever widening latency gap between memory and arithmetic operations [14] staggered increases in clock rates. Pollack's rule argues that processor performance increases are roughly proportional to the square root of the die size [15].

To continue reaping the benefits of Moore's law to provide higher performance, chip manufacturers turned to thread level parallelism [16]. This first manifested with thread Simultaneous MultiThreading (SMT) approaches that appeared in the IBM POWER5 [17], Ultra SPARC T2 [18], Hyperthreading [19] in the Pentium 4. Thread multiplexing allows multiple software threads to simultaneously share resources on a core. This technique is highly effective at hiding memory latency since one thread can be performing arithmetic operations while another is loading. This technique still exists on today's Intel processors and on GPUs, and is less expensive in terms of transistors than creating additional cores, but can quickly bottleneck with resource contention. As such, chip manufacturers have moved to duplicating processors on die to create multi-core chips.

Multi-core processors offer Symmetric Multi-Processing (SMP) performance on a single chip. Because of its benefits, multi-core technology is becoming pervasive, existing in markets ranging from high-end chips such as IBM's z196 mainframe processors [20] to Intel's Atom [21] processor designed for low power applications such as netbooks and cellphones. Multi-core processors overcome power density limitations by operating at lower frequencies while providing thread level parallelism. Most multi-core processors today have jack-of-all trades homogeneous cores, with a few notable exceptions.

While most processors in wide use today are homogeneous, the Cell processor is an important (now historical) exception. Featuring a single Power Processing Element (PPE) based on the PowerPC and eight Synergistic Processing Elements (SPEs) connected through the Element Interconnect Bus (EIB), the Cell processor provides very high single-precision throughput, weighing in at 230.4 GFlops [22]. Each SPE operates on 128 different vectorized registers and can quickly load and store in a deterministic number of cycles to its 256kB local store. Unfortunately, programming the Cell and tuning it presented numerous challenges: two compiler toolchains (one for the SPEs and one for the PPEs), managing data transfers, and efficiently branching in a vector-only ISA to name a few. While the Cell achieved moderate success, appearing in Sony's Playstation 3, IBM recently cancelled its development [23]. However, this only marked the start of the current heterogeneous computing era.

Academics and researches now use Graphics Processing Units (GPUs) pervasively in computational science as they are programmable enough to do far more than render graphics. Video accelerators have existed since at least 1982, when Intel introduced the iSBX 275 [24]. Sporting 32kB of memory (used to store the framebuffer), the iSBX could display up to 8 colors. The CPU issued simple commands to manipulate the pixels stored at specified memory addresses. In the mid-90s, several vendors such as ATI and 3dfx introduced a plethora of 3D accelerators targeting games and CAD applications. These GPUs were crude by today's standard, as developers could merely send geometry and textures to the device and let it render the scene. However, Nvidia's GeForce 3 [25] introduced programmable shading and the Radeon 9700 [26] was the first DirectX 9.0 compliant (compliance mandates programmable shaders with specified capabilities) accelerator. For the first time, researchers could use GPUs for things other than graphics.

Since the advent of programmable shaders, General Purpose GPU (GPGPU) computing has exploded. Early general-purpose computations using GPUs included fairly simple linear algebra operations that effectively amounted to tricking the device into doing math. For example, a 2003 paper describes using DirectX 9.0 and pixel shaders to compute matrix multiplication by sampling textures [27]. Early programmable GPUs had limited use in scientific applications due to their non-IEEE compliant floating point arithmetic and the fact that OpenGL and DirectX graphics calls drove the computation. However, their performance was compelling enough to encourage future research.

Enhanced GPU features along with new programming models increased interest in using video cards as computational accelerators. Brook [28] and CUDA [29] were among the first general purpose programming environments for GPUs. CUDA quickly fell into favor in the scientific community; developers began using CUDA in a wide range of applications ranging from linear algebra [30][31], chemistry [32], biology [33], and fluid simulations [34]. ATI introduced Close to Metal [35] and subsequently released their proprietary Brook+ and CAL programming languages. These programming environments saw some use [32][33].

With the birth of OpenCL [36], the era of proprietary GPGPU programming environments may be in its twilight.

The OpenCL platform is poised to become the leading cross-platform development environment for high performance computing within a single machine. Spearheaded by a number of vendors including Apple, AMD, NVIDIA, Intel, and IBM, OpenCL seeks to provide a single programming environment targeting multi-core processors, GPUs, and other accelerators such as FPGAs [37][38]. Already, researchers and developers are using OpenCL to develop cross-platform applications that take advantage of accelerators and multi-core processors [39][40].

OpenCL is even more interesting in that its device model allows for heterogeneous computation. It exposes different vendor implementations as platforms, allowing applications to simultaneously use devices from multiple vendors. For example, OpenCL can simultaneously use an Intel processor, an Nvidia GPU, and an ATI GPU in the same machine even though each company provides their own OpenCL library. Furthermore, a given implementation may support multiple device types; AMD's OpenCL offering supports AMD GPUs, x86 CPUs, and their Accelerated Processing Unit (APU) line. With the correct combination of vendor drivers, OpenCL can expose every computational resource available in a user's computer.

1.2 Problem Statement/Background

Effectively using heterogeneous computing resources is a challenge, even with a common language and environment such as OpenCL. Some applications [41][42] use resources to compute only tasks that map well to that resource. This works well when one computational resource is far quicker at computing some task than others. Programmers usually decide which tasks to map to which resources and how to move data. This often leads to static task distribution, where the developer assigns tasks to a specific device type at compile time. In [43], the authors dynamically scheduled tasks on multiple accelerators, but still

restricted task scheduling based on their type affinity to the underlying resource. When the performance discrepancy between resources for a given task isn't orders of magnitude and that task dominates the runtime, dynamic heterogeneous scheduling makes sense.

Applications that feature a runtime dominated by a single task-type with a moderate amount of device-performance agnosticism may benefit from using every resource in a system. To give a compelling example, suppose an application takes 1000 time units to execute where 99% occurs in a multitude of tasks of a single type. This type of profile often occurs in embarrassingly parallel problems. Further suppose that device 1 can compute 4 tasks per time unit and device 2 can calculate 1 task per time unit. Further suppose that the machine has one of each device. Using only device 1, the job requires 257.5 time units according to Amdahl's law [44]. Using only device 2, the job takes 1000 time units. Using both devices with optimal load balancing, the job takes 208 time units, a 1.24x speedup over just using device 1; the aggregated throughput of both devices is nearly 5 tasks per time unit.

1.3 Proposed Problem: A Parallel For Loop on Heterogeneous Resources

The core of the efforts in this dissertation is how to efficiently schedule tasks with data-dependent performance on heterogeneous resources. Specifically, one can state this problem as "how many and which iterations of a parallel-for loop should a scheduler give to each device in making forward progress in the program?" Answering this question requires addressing a number of challenges related to the problem statement.

Firstly, devices each have their own performance characteristics. The underlying architecture in each device exposes different cache configurations, high-speed memory, compute cores, clock rates, and other features that affect kernel execution rates. For example, adding vectors on a GPU is likely faster than on a CPU because it has higher aggregate

memory bandwidth. However, highly sequential kernels will execute more quickly on a CPU because of their higher clock rates and more aggressive instruction level parallelism.

Software kernels can also exploit hardware features in a particular device. For example, a kernel may use images and swizzling to exploit hardware acceleration on a GPU, while another kernel may load and store exclusively to global memory since a CPU caches these operations. In practice, developers can make tuned kernels for each device in their system. Tuning kernels to specific architectures allows one to squeeze the most performance out of them, but in the process can exacerbate architectural discrepancies as kernels' performance reflects the underlying hardware.

Task granularity plays an important role in scheduling iterations on a collection of devices, homogeneous or heterogeneous. From a scheduling and load balancing perspective, scheduling each iteration independently should yield more optimal load balancing because no device ever pulls more iterations than it needs and starvation never occurs. However, this creates enormous scheduling overhead when the task granularity is anything but large. Accelerators further compound this issue since they themselves expose finer grain parallelism that may need multiple iterations to fully exploit. In an execution model such as OpenCL or CUDA where the developer defines an execution workspace, users can encode the scheduled iterations into the work items/grid to execute in parallel. If each task takes the same amount of time, this approach still has a fairly trivial solution: assign iterations in proportion to their performance capability. For example, if device 1 can always compute 5 tasks per time unit and device 2 can always compute 2 tasks per time unit, giving 5/7 of the tasks to device 1 and 2/7 of the tasks to device 2 yields an optimal schedule. However, when tasks have less predictable runtimes, this solution quickly breaks down.

Many applications have data dependent performance; some iterations in a loop in these applications take longer than others because of additional processing needed on their inputs. In this case, static scheduling can falter, especially on heterogeneous architectures. If a slower device pulls iterations that happen to require more time because of the associated data, faster devices may starve. To overcome this problem, we propose that the scheduler

try to model execution time for iterations to make better informed decisions and execute the appropriate number of iterations.

To address the problems associated with scheduling loop iterations onto heterogeneous resources, we introduce the PINA (PINA Is Not Acronymic) loop model. PINA is a self-scheduling algorithm that uses online performance data combined with offline autotuning to distribute work. The offline autotuning serves to provide each device with an appropriate number of tasks while the online modeling allows each device to more efficiently choose blocks of iterations.

Chapter 2

Literature survey

Scheduling tasks onto resources is a well studied problem with a rich collection of prior art. In 1988, Casavant and Kuhl laid out a taxonomy describing various scheduling algorithms [45]. An important distinction in their taxonomy is between static and dynamic scheduling. In heterogeneous computing, researchers have attempted both.

Many modern compilers can parallelize *for* loops to run more efficiently in a homogeneous environment. Unfortunately, the performance gap between compiled code and hand-tuned code increases with newer architectures [46]; users can rely increasingly less on compilers to generate performant code. Polyhedral models [46] are becoming a popular abstraction for taking advantage of ILP, SIMD, caches, and multithreading to execute loops more quickly. These models allow compilers to explore a variety of transformations and make algorithmic tradeoffs between often competing architectural traits. GCC can compile with the Graphite loop optimization library [47], allowing the compiler to to automatically parallelize loops for SIMD and multi-core processors while performing cache tiling.

In addition to compilers, a multitude of libraries exist that allow developers to parallelize applications. Unlike compilers that attempt to automatically generate parallel code, these libraries require user intervention to parallelize code. OpenMP [48] and Intel Threading Building Blocks (TBB) [49] both allow users to parallelize code, targeting homogeneous

SMP machines. Both OpenMP and TBB contain constructs for trivially exploiting loop parallelism as well as more sophisticated schemes. Message Passing Interface (MPI) [50] allows users to run code on distributed systems, including modern supercomputers. Using MPI is generally more difficult than TBB and OpenMP because developers must explicitly contend with data movement across a network.

Cierniak et al. explored a plethora of static loop scheduling techniques, including modified versions of those used with homogeneous systems [51]. They start with a modified version of static scheduling for homogeneous machines and expand their assumptions for the heterogeneous case.

Cierniak et al. take a three-pronged approach to heterogeneity: the loop iterations, processors, and network interconnects can all be homogeneous or heterogeneous. They denote homogeneous loops to mean that each iteration takes the same amount of time while heterogeneous means that each loop iteration has an affine function describing the amount of time it takes. The authors assume that this function is of the form $t_i = ai + b$ in the paper. Furthermore, while the authors don't explicitly state it, they also assume that the compiler or the programmer knows this affine function at compile time. This assumption is critical to efficient scheduling on homogeneous and heterogeneous devices [51].

Static scheduling is perhaps the most obvious way to distribute tasks to multiple processors. In this method, either the programmer or compiler assigns iterations to different processors at compile time in a blocked or cyclic fashion; if a loop has 10 iterations and a machine has 5 devices, each device executes 2 iterations. On homogeneous architectures, this method is the crudest and easiest to implement. Because each device determines at compile-time exactly which iterations to execute, scheduling overhead is minimal. However, oblivious static scheduling has the poorest load-balancing behavior for dynamic or mixed workloads; if each iteration doesn't take the same amount of time, some devices may finish earlier than others and undergo work starvation.

In principle, if the programmer knows the relative execution rates of each device and execution time is homogeneous across iterations, then dividing iterations according to

each device's performance contribution to the whole should yield a load-balanced schedule. Equation 2.1 shows how many iterations N device i should execute given D devices with execution rate R. Note the subtle assumption that devices' execution rates are iteration invariant. For example, suppose a loop has 100 iterations, device 1 can execute 1 iteration per time unit and device 2 can execute 4. Using equation 1, device 1 should execute 20 iterations and device 2 should execute 80. This will fully use the combined executional power of 5 iterations per time unit; both device 1 and device 2 will finish after 20 time units. When iteration execution time is heterogeneous, Cierniak et al. provide a way to transform the iterations into a homogeneous loop.

$$N_i = \frac{R_i}{\sum_{j=1}^{D} R_j} N \tag{2.1}$$

When loop iterations are heterogeneous as defined by Cierniak et al., one can combine loop iterations and form a new homogeneous loop [51]. Since the authors assume the time taken for iteration i is of the form $t_i = ai + b$, combining iteration i and $(N - i + 1)$ yields the same amount of time for all i as shown in equation 2.2. A transformed loop variable j that executes both the i and $(N - i + 1)$ iteration provides a homogeneous loop. A compiler can use this transformed loop with either a homogeneous or heterogeneous computer as previously described. The authors also take into account homogeneous and heterogeneous networks with and without contention. For many applications where communication is minimal, a model can merely lump network costs in with computation costs.

$$t_i + t_{n-i+1} = ai + b + a(n - i + 1) + b = a(n + 1) + 2b \tag{2.2}$$

Cierniak et al then run a number of applications including matrix multiplication, an economics application, and a synthetic heterogeneous loop using the techniques described in their paper [51]. Their results show improvement in every case, including near-ideal speedup for matrix multiply.

There are two main issues that prevent Cierniak et al's work from working effectively in the context of Specmaster. Firstly, as previously shown, the affine function that maps iterations to time varies across different datasets in Specmaster. Worse still, Specmaster cannot ascertain the affine function a priori for a given dataset. The second assumption that fails in Specmaster is that devices have constant relative speedups. Figure 6.5 shows that to the contrary, a GPU and CPU running Specmaster vary in relative performance as a function of both data and iteration.

Cheng et al applied dynamic self-scheduling to a parallel for loop in a grid computing environment [52]. They explored three different iteration chunking techniques: trapezoidal self-scheduling [53] (TSS), factoring self scheduling (FSS) [54], and guided self scheduling [55] (GSS). Their conclusions show that self-scheduling alone performs poorly in a distributed matrix multiply.

To compensate for this, the authors [52] hybridize static and self-scheduling. The variable α partitions the workspace into static and dynamic regions. Their algorithm statically schedules the first $\alpha\%$ of a loop's iterations as per equation 2.1 and dynamically schedules the remaining iterations using a number of different methods. In this case, the authors use the processors' clock frequencies as the performance metric. Cheng et al found that $\alpha = 80\%$ yields good performance for GSS, FSS, and TSS. Furthermore, they found FSS to be most invariant to the α parameter; execution times varied less than 5% as a function of α while simultaneously achieving the highest performance of the three chunking algorithms.

Unfortunately, the authors chose a well-behaved application to demonstrate the effectiveness of their findings. Dense matrix multiplication and other dense linear algebra routines have well-known performance characteristics that work well with static scheduling, even in a heterogeneous environment [41]. In fact, the authors found that increasing α to 80% yielded the best performance regardless of chunking method used. The authors could have shown the α=100% case in their work as a baseline for comparison. In applications with data-dependent performance, different alpha values could be optimal for different data sets. Furthermore,

statically scheduling workloads is difficult when the devices' relative performances are not a constant function of the data set and iteration (figure 6.5).

A number of other related papers use similar techniques to target heterogeneous clusters [56][57][58]. In [56], the authors apply the same α hybrid scheduling technique to compute matrix multiplication on an "extremely" heterogeneous cluster. They found that once again, setting α equal to 80 yielded significant performance gains over static scheduling (α=100). Furthermore, their GSS/80 (Guided Self Scheduling with α=80) algorithm outperformed pure GSS (α=0) by 30%.

In [58], Shih et al. introduce Hybrid Parallel Loop Scheduling (HPLS). This is nearly identical to the work in [56] with a few slight changes. Rather than use the CPU clock speed to measure each device's performance, the authors use each devices relative execution rates for a particular problem (matrix multiplication in their example). Also, the master node also performs work rather than just distributing it. This work found modest performance improvements over [56] and [52].

In [57], the authors additionally allow a multi-core master node to do work on non-scheduling cores. They introduce Layered Self Scheduling (LSS) to take advantage of shared memory on multi-core systems. This reduces communication overhead on shared memory systems. Finally, the authors modify the LSS algorithm to weigh chunks according to their performance. They call this approach Enhanced Layered Self Scheduling (ELSS). Their algorithm maintains an array of the relative performances of a problem running on each machine (matrix multiplication in their example) and uses the ranking information to determine which chunk a process should receive. For example, the third fastest CPU receives the third largest chunk available. This leaves larger chunks for faster machines. Wu et al. found that ELSS yielded an average 1.35x speedup over LSS in matrix multiply using both GSS and FSS chunking algorithms. Unfortunately, they don't compare their work to the α scheduling method.

Jiménez et al [59] propose using predictive runtime code scheduling. This technique falls more in line with the work presented in this dissertation. The authors have a two-fold

approach to scheduling tasks: 1) PE (processing element) selection and 2) task selection. The first piece uses a heuristic to determine where a task should run. The second piece determines which task a PE should run when it becomes idle.

For the first task, the authors contend that traditional methods such as running a task on the first available resource is a poor idea in a heterogeneous environment. If task A takes 20 time units on device 1 and 1 time unit on device 2, the scheduler can wait up to 19 time units for device 2 to become available, run it on device 2 and still break even. To form a basis for their work, they propose four algorithms: first free (FF), FF round robin, history-gpu, and estimate-hist [59].

The two first free algorithms are very similar and only differ in how they choose a device when none are free. The vanilla algorithm uses a function g to decide where to schedule such a task whereas the round robin version iterates through a weighted round-robin of devices. For example, with weights 4 and 1, the scheduler would assign four tasks to device 1 for every task on device 2.

The history-based schemes use performance estimation to schedule tasks. Both algorithms keep a performance history for each device and use this information to decide where to schedule a task. If a task's performance ratio of device A to device B is greater than some value θ, then the scheduler forbids that task from running on device B; it can only run on device A. Thus, both algorithms create a collection of allowed devices for a given task type. The schemes differ in how they choose which of the allowed devices a task uses.

History-gpu chooses the first available resource in the allowable set. If no such resource is free, the algorithm schedules a task to run on the GPU.

Estimate-hist attempts to schedule tasks onto an allowable device with the least amount of time in its queue. The authors don't go into detail of how this works, but presumably maintain a running average of times for each device/task pair. A more sophisticated method could keep a number of data points for different problem sizes and interpolate to predict the execution time.

The authors found that using a performance prediction model to schedule tasks improved speedup over FF algorithms when concurrently running 4 or 6 benchmarks. Furthermore, in some cases, FF provided slowdown while the history-based methods still provided speedup. Jiménez et al show that estimating runtimes to schedule tasks is a viable method for executing tasks on heterogeneous resources [59].

Another common scheduling technique is work stealing [60][61]. This approach allows threads to take work from other threads when they become idle, preventing starvation and creating load balancing. Unfortunately, this approach doesn't work well in the context of OpenCL or CUDA, as kernels are atomic; one device can't steal work from another device's OpenCL or CUDA work queue. A given can load balance its threads onto the multiprocessors in a given device using work stealing [62], but balancing multiple kernels on multiple devices remains difficult to impossible depending on the exact devices used and their capabilities.

Maestro[63] attempts to autotune OpenCL kernels for different heterogeneous devices and split work up automatically. This includes finding the optimal work-group size for each device, attempting to automatically overlap computations and data transfers, and load balancing. In the last component, they authors attempt to do this by characterizing each device using OpenCL API calls and benchmarks. They repeatedly measure kernel execution times running on different devices and compute a weighted average to statically distribute work.

Most previous work that focuses on heterogeneous scheduling make several assumptions that don't hold true in the general case. Firstly, devices don't necessarily have consistent relative performance even for the same operation; problem size and data can introduce variability into relative execution rates. Secondly, the scheduler may not know a priori what the execution time for a given task will be. In simple applications such as matrix multiplication, the scheduler can usually infer task completion time from its problem size, but with data dependent performance this may be impossible or too expensive to model from the data itself.

Table 2.1: Task scheduling taxonomy

Previous work	Task partitioning	Task scheduling	Relative performance knowledge
Cierniak et al[51]	static	static	heuristic
Self-scheduling[53]	static	dynamic	heuristic
Work stealing[61]	dynamic	dynamic	heuristic
Shih et al[58]	static	hybrid	heuristic
Wu et al[57]	static	dynamic	offline empirical
Jiménez et al[59]	static	dynamic	online empirical
Maestro[63]	autotuned static	autotuned static	both offline and online empirical
PINA	dynamic	dynamic	online empirical

Table 2.1 shows a classification of various scheduling algorithms including the proposed work. In terms of approach, the proposed work most closely aligns with Jiménez et al. Both works use online models to determine where to run tasks. However, in that work users submit discrete tasks and the scheduler merely assigned where they ran whereas the proposed work breaks a loop's iteration space into subtasks and schedules those. Furthermore, this work ties "problem size" into the equation with a simple model. In terms of goals, this work most closely relates to Shih et al and Wu et al. All three attempt to dynamically schedule loop iterations onto heterogeneous devices. However, the work in this dissertation makes different assumptions about the underlying workloads; in [57] and [58], the authors assume data-agnostic applications and that relative device performances can be obtained offline.

Chapter 3

clUtil*

clUtil uses C++11 features including classes, RValue references, variadic templates, and lambdas to provide highly readable and terse OpenCL frontend code. Figure 3.1 gives an example that uses all OpenCL devices (GPU, CPU, and otherwise) in a machine to add two vectors.

clUtil provides significant productivity improvements through the way it abstracts OpenCL. clUtil initializes all devices on all platforms into a single flat array, creates a context and two command queues (to allows users to overlap computation and data transfers). It then compiles a list of files passed by the programmer for each device and creates a kernel lookup table using an STL map. This allows users to reference kernels by name rather than managing cl_kernel handles. Users query and set the device to issue subsequent commands using Device::{Set, Get}CurrentDeviceNum()

*This chapter contains excerpts from "A Trip to Tahiti: Approaching a 5 TFlop SGEMM using 3 AMD GPUs," presented at SAAHPC 2012. I am the principle author of this text and the underlying work it represents. The other author, my advisor, serves as the principle investigator for the broader project encompassing this work.

```
//aDev, bDev, and cDev are preallocated arrays of pointers to
//clUtil::Buffer types with one of each buffer per device

ParallelFor(0, 1, vectorLength - 1, [&](size_t start, size_t end)
{
  unsigned int count = end - start + 1;

  size_t curDev = Device::GetCurrentDeviceNum();

  aDevice[curDev]->put(&a[startIdx], count * sizeof(float));
  bDevice[curDev]->put(&b[startIdx], count * sizeof(float));

  clUtilEnqueueKernel("vectorAdd",
                      clUtilGrid(count, 64),
                      *aDevice[curDev],
                      *bDevice[curDev],
                      *cDevice[curDev],
                      count);

  cDevice[curDev]->get(&c[startIdx], count * sizeof(float));
}, StaticSchedule());
```

Figure 3.1: Vector addition example using clUtil's ParallelFor(). vectorLength is assumed to be sufficiently short so as not to overflow buffers on the devices.

3.1 Memory management

clUtil effectively replaces all cl_mem handles with Buffer and Image objects. These objects both inherit from the abstract Memory class, which actually contains the underlying cl_mem reference. Image objects have three constructors for creating 1D, 2D, or 3D images. Since OpenCL 1.1 doesn't actually support 1D images, clUtil emulates them by mapping the 1D indices onto a 2D index set. clUtil provides functions for reading and writing to 1D images that automatically transform the coordinates on the user's behalf. This makes them simple replacements for large buffers on devices that don't support cached global memory in kernels that need fairly random access. Buffer objects have a single constructor that takes the size in bytes as its parameter. Buffer and Image objects allocate on the current clUtil device.

Users copy data to devices using the put() method and copy data from the device using get(). Memory objects track dependencies using their cl_event handle to the last call to get(), set(), or clUtilEnqueueKernel(). This allows all data transfer methods to issue asynchronously, allowing the main thread to continue enqueuing tasks for devices.

Memory objects implicitly use OpenCL's underlying reference counting for memory management. Its copy constructor calls clRetain() on the underlying cl_mem handle and its destructor calls clRelease(). Using this method, memory objects automatically deallocate when no references exist, reducing programmer burden in avoiding memory leaks.

3.2 Launching kernels

clUtil uses a variadic function to enqueue kernels on the current device. The first argument is the name of the kernel and the second is the grid defining the number of global work items and the number of work items per group. The clUtilGrid function is variadic, taking pairs of global and local work items in an arbitrary number of dimensions. It returns a variadic Grid class containing dimension information accessible through virtual functions. The returned class gets passed to clUtilEnqueueKernel() by RValue reference, obviating object handles

and creating a clean syntax for defining the execution grid. The remaining parameters map to the underlying kernel parameters and should match in type.

When processing the remaining parameters, the clUtilEnqueueKernel() function calls clSetKernelArg() directly on the argument with the argument index equaling the position in clUtilEnqueueKernel() minus two. One exception to this rule arises when the user passes Image objects, Buffer objects, or a pointer to a Memory object. In these cases, clUtilEnqueueKernel() appends the last event associated with the Memory object to a list of dependences and then passes the underlying cl_mem handle to clSetKernelArg() rather than the object itself. After setting all kernel arguments, clUtil passes the dependency list to clEnqueueNDRange() and updates all passed memory objects with the resultant cl_event.

3.3 ParallelFor

One of the design goals of clUtil is to easily allow composition of applications that can exploit multiple heterogeneous devices. To achieve this, we provide the *ParallelFor* loop. Users pass this function an index range, a lambda, and an optional scheduling algorithm and the loop executes the lamda using devices as they become available. To increase flexibility, we separate the event loop from the model that doles out work. This allows the user to specify one of (currently) three models depending on the workload and device configuration.

The static scheduling algorithm divides the index range into a number of equal sized chunks, allowing devices to take work as they become idle. This model works reasonably well in a homogeneous environment where workloads don't change much as a function of the loop index, such as the one presented in this paper (using only the 3 Radeon 7970s).

The EGSS (Enhanced Guided Self Scheduler) is an adaptation of the work found in [57]. It iteratively divides work into chunks as per the GSS algorithm and assigns the the k^{th} largest chunk to the k^{th} fastest device group. EGSS uses a heuristic based on the number

of processors found in the device to determine the device performance ranking ($16 * CU$ for GPUs and $1 * CU$ for CPUs, where CU is the number of compute units on the device).

Finally, the PINA Scheduler is the raison d'être for this dissertation (Chapter 4). It combines offline autotuning with online performance sampling to schedule work on available devices.

When calling ParallelFor(), the main thread enters an event loop that terminates when all iterations complete (source code given in Appendix A). All devices are initialized as idle. Upon detecting an idle device, the loop queries the scheduling algorithm if said device has work available. If so, the event loop gets the index chunk from the model, starts a timer, changes the current device, calls the user lambda, and finally enqueues a series of markers to get cl_event so the event loop can determine when the devices completes all tasks in its queues. After issuing new tasks to idle devices, the event loop polls for completed marker events.

Upon seeing a completed set of loop iterations, the event loop stops the timer associated with that device and notifies the model which device completed its iterations and how long it took (which the model is free to ignore). The event loop then marks the device as idle and decrements the number of iterations remaining. Using this event loop rather than threads leads to a dramatically simplified implementation, but creates the requirement that the user lambda cannot block in any way, lest the loop serializes. All get() and put() methods as well as clUtilEnqueueKernel() execute asynchronously with this caveat in mind. When the main thread returns from ParallelFor, it guarantees that all iterations have completed.

3.4 Profiling

One final feature of clUtil is the ability to enable event profiling. When enabled, profiling allows users to start and stop profiling using function calls and visually view all data transfers and kernels as a function of time in an output Scalable Vector Graphics (SVG) file. This

feature merely piggybacks off clUtil's dependency tracking mechanism. When calling get(), put(), or clUtilEnqueueKernel(), clUtil retains the underlying tracking events. When the user calls DumpProfilingData(), clUtil generates an SVG file from the retained events' start and end times as well as their event type.

Chapter 4

PINA Scheduler

4.1 Assumptions

The primary work in this dissertation focuses on scheduling parallel loop iterations on heterogeneous devices. This requires several assumptions deduced from data gathered from Specmaster.

This work relaxes several assumptions over previous work including:

1. Each loop iteration is independent of all others.
2. The time each iteration takes is a function f_d of the device on which it executes (d), the iterator value, and underlying data.
3. The runtime needn't know f_d a priori.
4. Interpolation yields a reasonable approximation to f_d.
5. $f_i = f_j$ if device i and j are the same type of device. For example, two Radeon 5870s should have the same execution rates over the iteration space.
6. Devices' relative performances p vary as a function of the iterator value and underlying data.

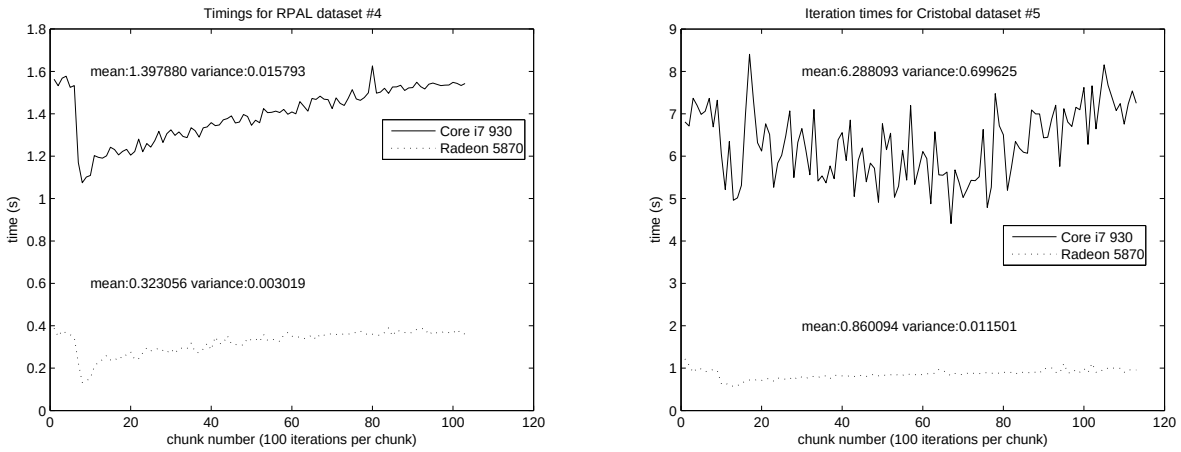

Figure 4.1: Iteration execution time on a Core i7 930 and Radeon 5870

7. Caching has little effect on performance with respect to the order of iteration execution.

8. Devices have higher throughput with larger iteration chunks (a contiguous group of iterations) rather than smaller. The function relating performance to chunk is of the form $1 - exp(\alpha_d s)$ where $\alpha_d < 0$, s is the chunk size, and d is the device. Furthermore, this function is roughly constant regardless of the underlying data, since it represents the program's ability to amortize loop iteration costs.

Figure 4.1 shows that two different data sets yield different execution profiles in the same program on the same device. This fact gives rise to the first two assumptions. f_d differs in both graphs in shape, implying we may not know what the function will look like before we run the loop. Furthermore, the shapes if the graph functions are certainly not of a consistent form, meaning f_d could be nearly any function of index and data.

Previous work makes fairly optimistic assumptions about f_d or tries to ignore it. [51] assumed that $f = ax + b$, as it allowed them to statically combine large iterations with small ones to create a homogeneous loop. By then assuming the speedup of device d_i over device d_j is a constant function of the loop index, they could then statically schedule iterations optimally as per their assumptions. Conversely, the self-scheduling techniques in

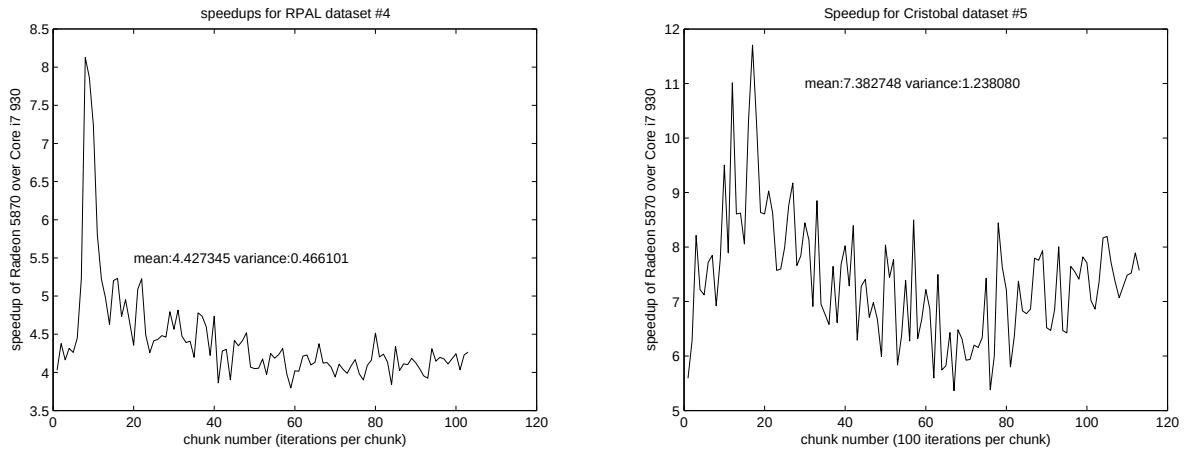

Figure 4.2: Speedups of Radeon 5870 over Core i7

[52][56][58][59] attempt to dodge the issue by allowing dynamic scheduling to load balance the iterations. However, each chunking technique has some pathological f that will completely break their load balancing attempts. For example, if a non-trivial chunk requires more time than the sum of the remaining iterations, load balancing will fail. Breaking this chunk into smaller pieces would alleviate this issue.

The works discussed in the literature review assume f_d is known if it isn't ignored. [51] implicitly assume that the compiler knows f_d by assuming f_d is a linear function. Conversely, this work acknowledges the fact that this function may not be known at compile time, or even pre-loop runtime.

Figure 4.2 shows that devices' relative performance is not necessarily constant. This complicates an optimal static schedule defined in equation 2.1; devices execute some iterations relatively more quickly than others. Intuitively, a scheduler should give iterations where f_{d1}/f_{d2} is high to d_1 and iterations where this is low to d_2 assuming it can still load balance. In machines with more devices, this problem becomes more complicated.

Another factor impacting runtime is scheduler and runtime overhead. Theoretically, a scheduler could issue single iterations at a time to devices. This method is known as Pure Self Scheduling (PSS) [57] and offers extremely good load-balancing. Unfortunately,

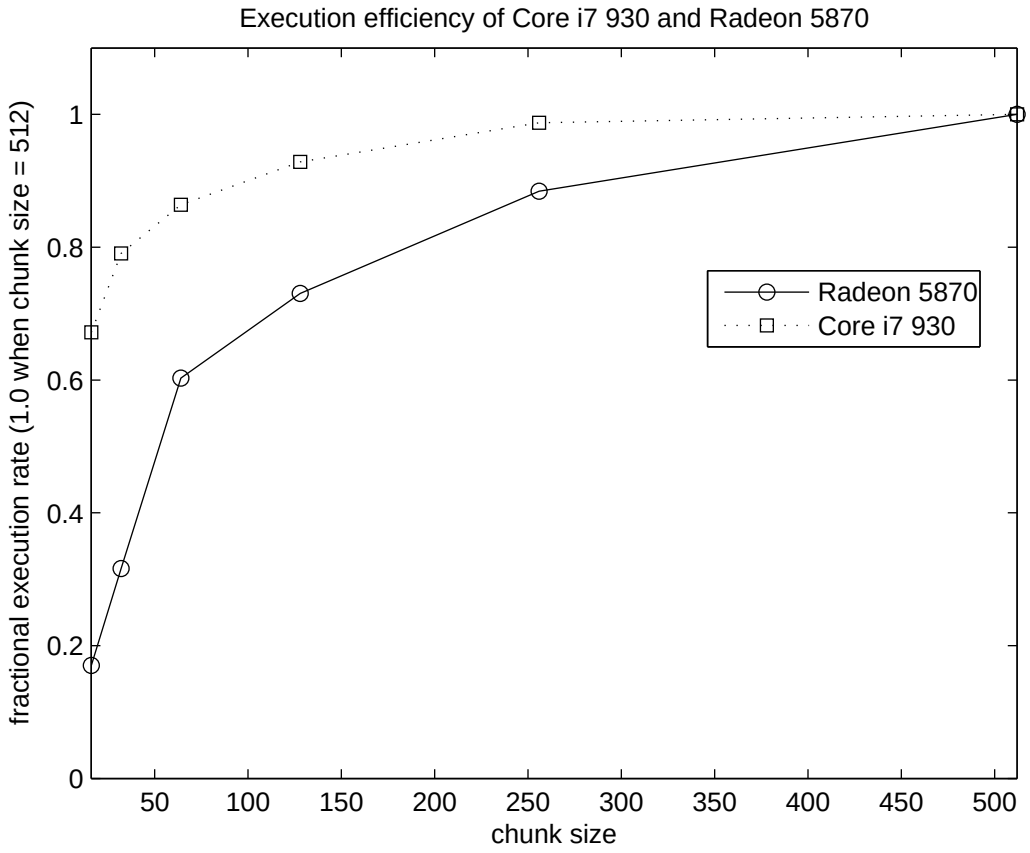

Figure 4.3: Fractional execution rate on D04_Cristobal dataset number 5

worker devices must invoke scheduling overheads each time they do a single loop iteration. Furthermore, in a GPU kernel where a range of loop indices maps to the execution grid, the device may be highly underutilized.

Figure 4.3 shows Specmaster's execution rate on D04_Cristobal dataset number 5. In this experiment, we pulled different sized chunks and passed each chunk to an OpenCL kernel for processing, timing the cumulative runtime of all iterations; smaller chunks yields more kernel calls while larger chunks yield fewer. The GPU's performance is highly sensitive to the number of iterations it executes simultaneously, an artifact of its data-parallel design. This indicates that a well-designed scheduler should generally provide large chunks of iterations to a device while, possibly making trade-offs with load-balancing to do so.

Equation 4.1 explains why larger chunks give higher execution efficiency on devices. Since t is a function that increases with chunk size, it begins to dominate the equation as the chunk size increases. When the chunk size is small, execution overheads (including kernel launch, driver and kernel calls) dominate the runtime as they aren't a function of chunk size. This effect is compounded on parallel architectures, since small chunk sizes may result in an underutilized device.

$$t = t_{overhead} + t(chunk\_size) \tag{4.1}$$

The challenge in creating an efficient heterogeneous scheduler is then to address the previously mentioned issues.

4.2 PINA: an Efficient Scheduler for Heterogeneous Workloads

PINA is a parallel for loop scheduler designed for heterogeneous workloads running on heterogeneous devices. It combines online performance metrics and offline autotuning to answer a parallel for loop's essential question: "Which and how many iterations should this loop distribute to a given idle device?" More succinctly as two questions, "which iterations?" and "how many?" PINA addresses the former question by sampling the search space and approximating f_d. It answers "how many" through offline autotuning to find chunk sizes that amortize kernel and transfer overheads.

4.2.1 Choosing Which Iterations

The novel aspect of this work is to approximate f_d online for each device d online and use this information to schedule iterations. The scheduler can approximate f_d by running iterations at regular intervals and measuring their execution times (e.g. sampling f_d). Equation 4.2

shows the lowest sampling rate needed to accurately reconstruct a signal (e.g. the Nyquist rate [64]) to avoid aliasing. B is the signal's bandwidth, the highest frequency with non-zero power. However, since the application doesn't know f_d a priori, we can't make any claims to under or oversampling. While this may disallow perfect reconstruction of f_d, in practice, the scheduler should only need a reasonable approximation to make more sophisticated scheduling decisions.

$$f_{NYQUIST} = 2B \tag{4.2}$$

The scheduler approximates f_d for each device type. If more than one device of a given type exist, then they can divide the sampling process. Since the sampling process takes longer on slower devices, any proposed algorithm must either insert a barrier to wait for the sampling process to complete or make scheduling decisions with an incomplete model. While the former is easier to implement and can prevent the scheduler from making ill-informed decisions, the latter maintains seamless execution.

The number of samples each device takes is another algorithmic trade-off. More samples yields a more accurate model the scheduler can use to make decisions. However, more time spent sampling means less time actually using the model. As the number of samples increases, samples can break up iteration chunks which would prevent the scheduler from packing large workloads to devices. Furthermore, spending too much time sampling may cause slower devices to never make informed decisions on which iterations to execute. Finally, the scheduler can further refine the model with runtimes of actual scheduled iterations after the sampling phase, possibly obviating the benefit of a large number of samples.

With the approximations for each f_d, the scheduler can proceed issuing iterations to devices. This entails assigning execution iterations chunks C_1, C_2, etc. to devices. Each device d consists of a union of these iterations chunks (equation 4.3) with additional constraints. No two of D total devices execute the same iteration (equation 4.4) and all devices together execute all iterations (R) in the loop (equation 4.5). Both of these

constraints apply during the sampling phase as well; chunks executed while sampling f_d reside in R_d.

$$R_d = C_i \cup C_j \cup ... \tag{4.3}$$

$$\bigcap_{i=1}^{D} R_i = \emptyset \tag{4.4}$$

$$\bigcup_{i=1}^{D} R_i = R \tag{4.5}$$

The overall goal of the scheduler is to assign workloads to devices in such a way that minimizes execution time. Mathematically, this entails constructing a piecewise function $g_{optimal}$ by choosing each R_d that minimizes g (equations 4.6 and 4.7). Unfortunately, this problem is a more generalized form of the multi-processor task scheduling problem and is NP-complete [65]. To demonstrate that this is a generalization, let $f_1 = f_2 = ... = f_D$, which is the symmetric multiprocessor case. As such, a practical scheduler can only approximate $g_{optimal}$ with a heuristic.

$$g(i) = \begin{cases} f_1(i), i \in R_1 \\ f_2(i), i \in R_2 \\ ... \\ f_D(i), i \in R_D \end{cases} \tag{4.6}$$

$$g_{optimal} = min(\sum_{R} g(i)) \tag{4.7}$$

Static scheduling with runtime approximations for each f_d presents a number of challenges. Firstly, statically assigning tasks requires fairly complete information about task runtimes. This implies that the loop should barrier synchronize all devices after modeling to ensure the

model is up to date and complete, a costly operation. Secondly, any practical assignment algorithm must use a heuristic to avoid solving an NP-complete problem. To avoid these challenges, we plan to use a dynamic scheduling approach.

Effective dynamic scheduling presents fewer challenges than assigning tasks statically. If some devices finish early, the scheduler can begin assigning them tasks with partial information about the model. For example, if the GPUs have finished their 10 samples of f_{GPU} while the CPUs have only completed 2 samples of f_{CPU}, the scheduler can interpolate from only those two samples to produce a very crude model that becomes more refined as the CPUs complete more samples. Alternatively, the GPU can issue tasks only in regions for which all devices have sampling information. This immediately raises the question of how an algorithm should sample f_d.

Figure 4.4 shows two possible methods for acquiring samples of f_d. Linear sampling progressively provides an accurate approximation of a portion of f_d while tree sampling provides a rough estimate of all of f_d and then refines the approximation with more samples. The former yields dramatically simpler implementation and lower overhead. The latter could yield better results in workloads with little variance, but provides inaccurate information when f_d changes significantly between sampling regions.

Once a given device has sampled iteration space and becomes idle, the scheduler must answer two questions in assigning it iterations: which and how many. Both questions have a number of practical concerns that can serve as competing ends. Optimally answering the "which" question at some point in time may fragment the iteration space leading to small chunks that hurt execution times later. Conversely, choosing too many iterations may better amortize scheduling overhead at the expense of load balancing.

Fragmentation is a possible concern when answering the "which" question (figure 4.5). The iteration space gets broken into disjoint pieces as a result of the sampling process. Using more samples may provide a more accurate model at the cost of increased fragmentation. Fragmentation is undesirable for two reasons: the scheduler requires a larger and possibly

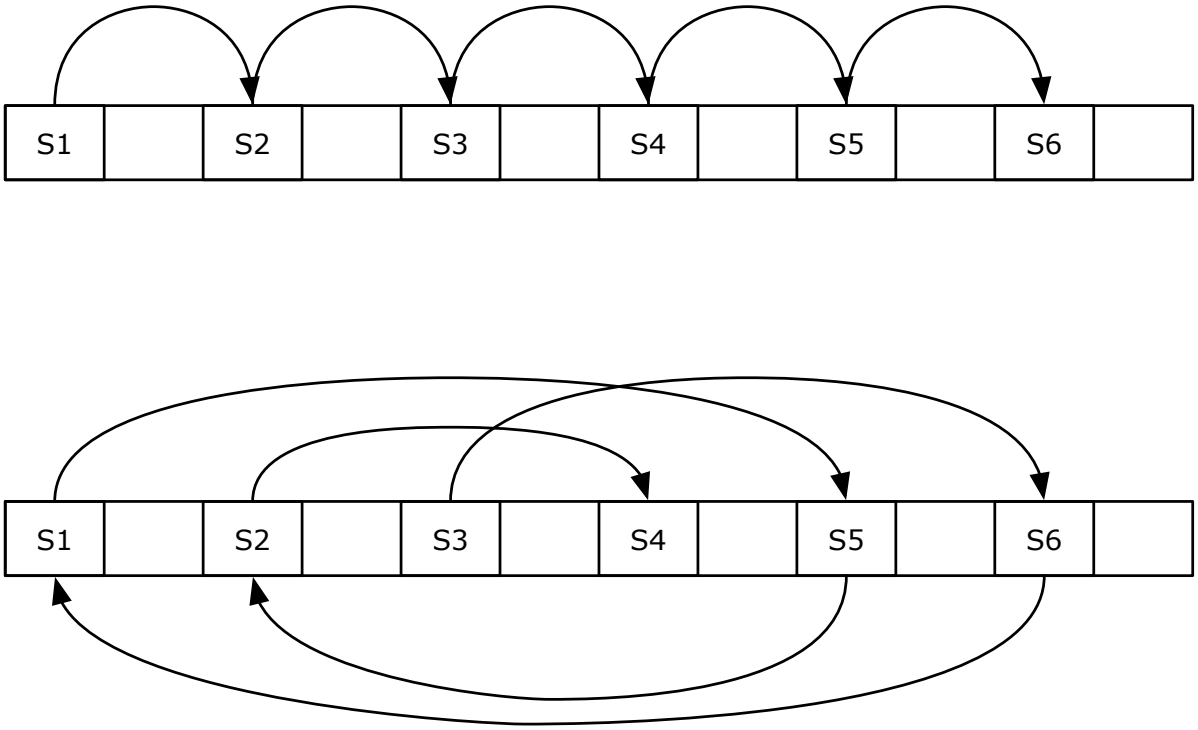

Figure 4.4: Linear (top) and tree (bottom) sample iteration

Figure 4.5: Fragmentation can occur when the scheduler makes unconstrained scheduling decisions

more complex data structure to scoreboard which iterations remain and smaller chunks prevent devices from amortizing scheduler overhead (figure 4.3).

While some fragmentation is unavoidable due to sampling, adding constraints as to which iterations a device can select at a given time can reduce fragmentation during the normal execution phase. More specifically, by requiring that all proposed chunks be adjacent and following either a sample or previously executed chunk, the scheduler guarantees that no more fragmentation occurs than exists after the sampling phase; the number of holes never increases (figure 4.6). Disallowing a chunk to execute adjacently preceding a sample dramatically improves implementation ease by simplifying the required scoreboarding and sampling.

4.2.2 Choosing the Number of Iterations

The "how many" question has overhead amortization and load balancing as competing ends. Giving a device too few iterations introduces significant scheduling overhead. On the other hand, issuing too many iterations to one device can cause others to starve. Many self-scheduling techniques address this issue by providing large chunks that execute with minimal overhead as well as smaller chunks to load balance. We propose using another method to solve this issue.

Gustafson's law[66] (equation 4.8) explains why devices perform better with larger chunk sizes and serves as the basis for an amortization model. Increasing the chunk size amounts to increasing $t_{parallel}$ while holding P and t_{serial} constant. Thus, the second term representing

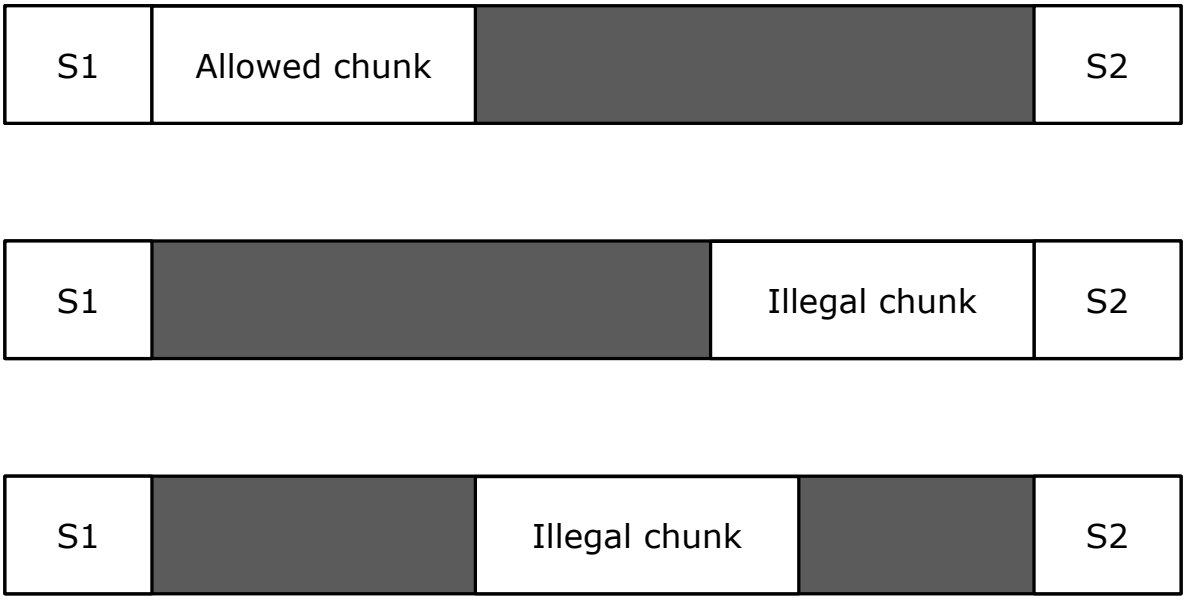

Figure 4.6: The scheduler may choose chunks adjacent to an after a sample (top), but not before a sample (middle), or anywhere else (bottom). This helps prevent fragmentation and simplifies implementation.

the parallel workload begins to dominate the total time. Unfortunately, parameterizing this equation is difficult to do empirically in this context. The serial time is a function of both kernel launch overhead and iteration time since iterations are the measured unit of parallelism. Furthermore, the what constitutes a "processor" according to Gustafson's law is becoming increasingly vague in modern architectures featuring thread multiplexing schemes where multiple threads share hardware to hide pipeline and memory latencies.

$$\begin{aligned} a &= \frac{t_{serial}}{t_{serial} + t_{parallel}} \\ S(t_{parallel}) &= a + P(1 - a) \end{aligned} \tag{4.8}$$

Due to the difficulties in parameterizing Gustafson's law in this context, we propose an approximate model. Normalizing Gustafson's law (equation 4.9) to yield relative speedup simplifies the model by bounding the relative performance between 0 and 1. When $P \gg 1$, $S(0) \approx 0$ and $S(\infty) = 1$. Furthermore, equation 4.9 is smooth on the interval $[1, \infty)$ and quickly approaches its asymptote of 1 and is monotonic.

$$S(t_{parallel}) = \frac{a}{P} + 1 - a \tag{4.9}$$

Equation 4.10 gives an approximation to Gustafson's law that accepts a single parameter. This serves as a reasonable approximation because $S(0) = 0$, $S(\infty) = 1$, it's monotonic, and quickly approaches its asymptote. Furthermore, varying the sole α parameter changes the convergence rate. Figure 4.7 compares Gustafson's law with this approximation.

$$S(t_{parallel}) = 1 - \exp(\alpha t_{parallel}) \tag{4.10}$$

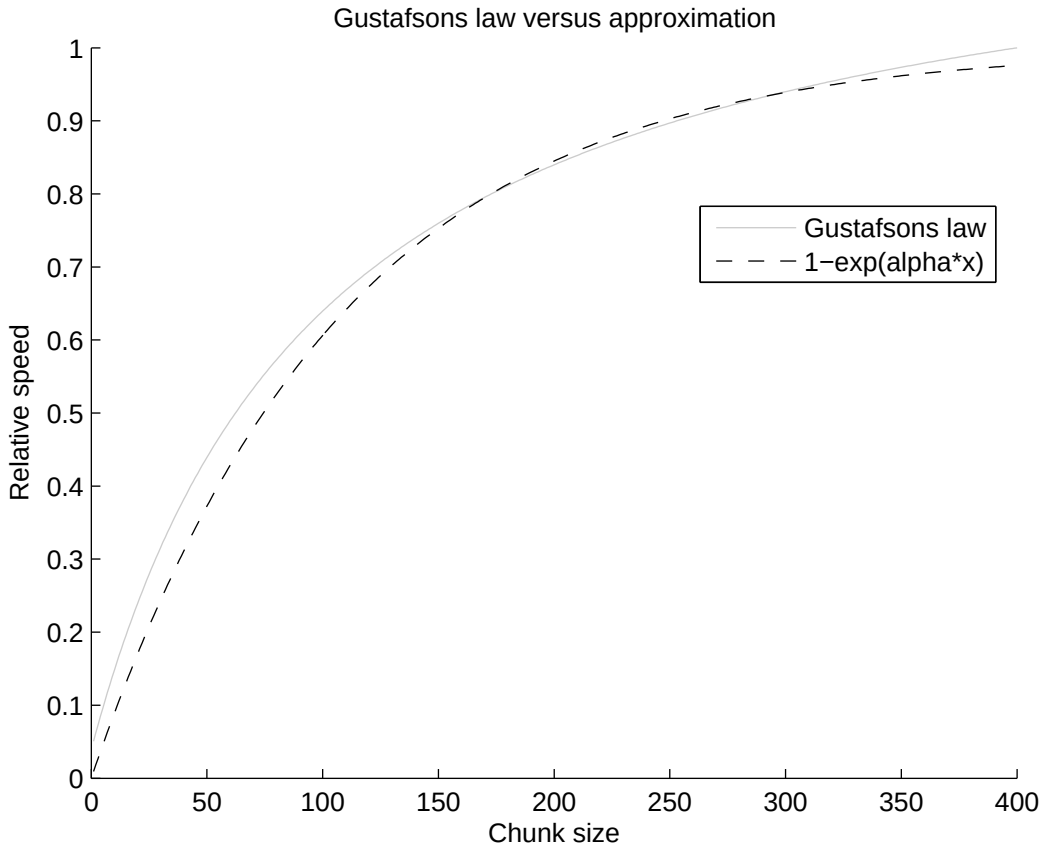

Figure 4.7: Gustafson's law with $t_{serial} = 100$, $t_{parallel} = 400$, $P = 32$ versus approximation

Assuming that parallel execution time and number of iterations are interchangeable, the model can directly predict the expected execution rate for a given chunk size (equation 4.12). Solving for some fraction $0 < \beta < 1$ yields the chunk size required to attain high performance. For a given for loop, this equation need only be solved once and the iteration and time information can be reused. In fact, this can be performed once in an autotuning step.

$$1 - \exp(\alpha_d chunk_{amortized}) = \beta \tag{4.11}$$

$$chunk_{amortized} = \frac{\log(1-\beta)}{\alpha_d} \tag{4.12}$$

To actually find α from empirical data, we employ a simple linear regression (equation 4.13). This approximation has a number of caveats. Firstly, when normalizing actual performance data, the $\max(S(x)) = 1$. This results in taking the natural log of 0 at the point where performance is highest, resulting in $-\infty$. To avoid actually hitting the asymptote in the normalized performance, we introduce a small ϵ term. This results in nominal deviation from the actual y'. The second caveat is that long tails in empirical data decrease the accuracy of the approximation. One way to overcome this is to remove data close to the asymptote (i.e. greater than 0.95). Figure 4.8 shows the effects of using and removing a long tail.

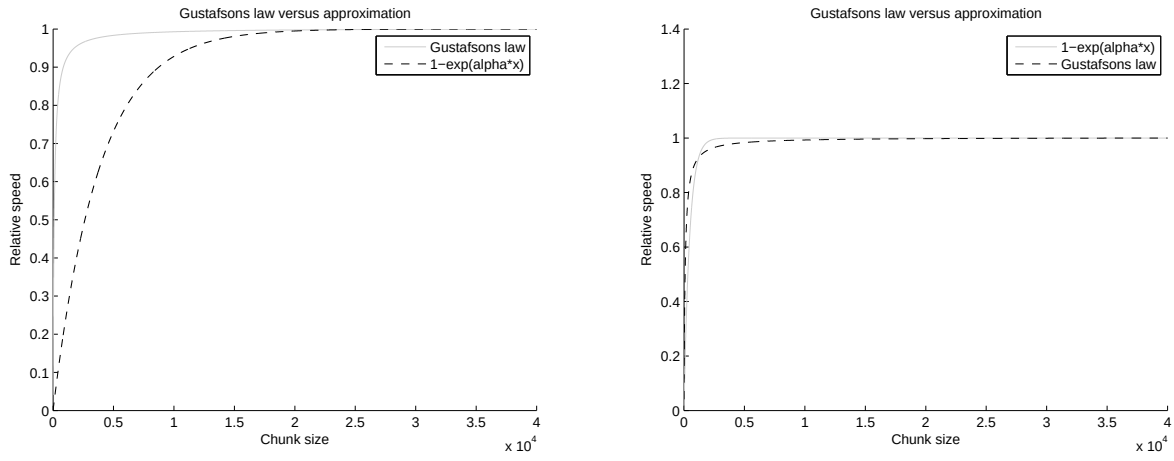

Figure 4.8: (Left) Approximation made with long tail values > 0.95 (Right) Regression performed without tail

$$
\begin{aligned}
x &= ||chunk|| \\
S(x) &= 1 - \exp(\alpha_d x) \\
\alpha_d x &= \log(1 - S(x)) \\
y' &:= \log((1+\epsilon) - S(x)) \approx \log(1 - S(x)) \\
y' &\approx \alpha_d x \\
\alpha_d &\approx (\overline{xy'}/\overline{x^2})
\end{aligned}
\tag{4.13}
$$

This work employs an autotuning technique to discover the ideal chunk sizes for each device type. Each time the PINA scheduler executes for a loop, it attempts to load autotuning data from disc containing execution rates sampled at powers of two for each device. If it fails to find any data for a given device type, then the loop hasn't been autotuned for that device type and starts autotuning with a chunk size of 1. If the PINA scheduler finds incomplete data, the scheduler fixes the chunk size to the next power of two and runs the loop using only the devices in the selected device group. In either case, the PINA scheduler records the sample data in a file associated with that device type and loop id. Sampling

data is considered complete when the last sampled chunk size yields less than some fractional performance improvement over the previous chunk size. When the sampling data completes, the scheduler discards the final sample and renames the data file to indicate completion. The PINA scheduler continues this autotuning process for each device type in the system.

Once the PINA scheduler has data relating chunk size to relative performance, it uses this information to discover good chunk sizes for each device. It normalizes the absolute execution rates of the loops loaded from file to the maximum execution rate (by construction, the last sample). It then performs a simple linear regression to find α_d for each device (equations 4.13), and solves for $chunk_{amortized}$ (equation 4.12).

4.2.3 Putting It All Together

Appendix B lists the function that assigns work to a given device. The function first checks if the specified device group has completed all of its samples. If not, it assigns work in the next non-full sampling region. If the device group has all its required samples, the PINA scheduler identifies the region where the device group has the highest performance relative to other device groups. It does this by computing the two norm speedup of the specified device group over every other device group interpolated at the first iteration in the candidate hole. Once the PINA scheduler selects a target iteration location, it then assigns either the desired work for that device group or the remaining work in the selected hole, whichever is lesser. The PINA scheduler then updates its list of available work and returns the assigned work.

4.3 Applications

To evaluate the performance of the PINA loop scheduler, we examine its performance compared to two other loop scheduling techniques: self scheduling with a fixed chunk size and an enhanced guided self-scheduling algorithm based off [57]. The latter loop scheduler implicitly places the second layer of scheduling within the OpenCL kernel instances. We evaluate the performance of each scheduler in three applications that represent three different workloads: matrix multiplication, peptide identification, and raytracing.

Matrix multiplication (chapter 5) represents the least difficult workload to efficiently schedule. There should exist virtually no performance variance with respect to loop iteration. This is because each iteration always performs the same amount of work and the execution rate in principle doesn't depend on the contents of the input matrices (barring some exceptions related to floating point arithmetic such as denormalized numbers). In principle, the two loop schedulers designed for heterogeneous

The second application, peptide identification (chapter 6), features a moderate amount of data-dependent performance. The performance of each iteration depends on how well its corresponding scan correlates to peptides found in the user given database. Scans that have precursor mass to charge ratios corresponding to fewer peptides take less time to process than those that correspond to more peptides. This caveat also yields device-dependent performance; processors execute more quickly on some iterations than others relative to GPUs.

The final application is a Raytracer (chapter 7). Raytracing is an ab initio method for rendering 3D graphics that features embarrassing parallelism. This application serves two purposes: firstly to demonstrate the ease in integrating clUtil's parallel for loop into an existing OpenCL application and secondly to examine heterogeneous execution in a strong scaling application.

Chapter 5

Matrix Multiplication*

Linear algebra and high-performance computing have a long and rich shared history. Since BLAS (Basic Linear Algebra Subprograms) debuted in the late 1970s[67], FORTRAN users have had a standardized package for performing linear algebra. BLAS originally cast all higher level operations in terms of lower level operations; a matrix multiplication was a series of matrix-vector products. However, as memory hierarchies changed due to the advent of caching, the level at which developers optimized routines increased.

Currently, one of the more important routines in BLAS is matrix multiplication. In principle, it's also one of the simplest consisting of 3 nested loops, an accumulate, and a store. Modern implementations however are rarely so elegant. Matrix multiplication is important because SGEMM (Single-precision GEneral Matrix Multiplication) and DGEMM (Double-precision GEneral Matrix Multiplication) serve as the building blocks for many higher level routines such as LAPACK's Cholesky and LU factorizations. If for no other reason, GEMM is still an important routine because it's integral to modern HPLinpack[68] benchmarks used to rank the world's fastest supercomputers[69] and topping it is a prestigious and expensive endeavor.

*This chapter contains excerpts from "A Trip to Tahiti: Approaching a 5 TFlop SGEMM using 3 AMD GPUs," submitted to SAAHPC 2012. I am the principle author of this text and the underlying work it represents. The other author, my advisor, serves as the principle investigator for the broader project encompassing this work.

GPUs have become increasingly programmable in the past decade. As a result, they now appear in some of the fastest computers in the world[70][71]. They offer an unrivaled combination of cost effectiveness, power efficiency, and programmability for many applications.

5.1 Previous Work

A number of applications have proven to be readily amenable to acceleration using GPGPUs. One of the earliest and notable examples is linear algebra. In [27], Moravánszky demonstrates matrix addition, assignment, and multiplication using Direct 3D. Given the state of general purpose programming models (i.e. nonexistent) for GPUs at the time, this work was necessarily crude and unwieldy by today's standards. Moravánszky essentially deceives the GPU into believing it's manipulating textures, addition being cast as image blending and multiplication being a series of vertex shaders. While the matrix multiply found in this paper was marginally slower than ATLAS[72] tuned for a Pentium 4 and wasn't IEEE compliant, work such as this set the stage for later advancements. Early work such as this demonstrated that GPUs held large potential for accelerating a variety of applications other than graphics.

In [30], Volkov and Demmel presented a performant matrix multiply running on a GTX 280 using CUDA. The authors found the GTX 280 to yield nearly a 5x improvement over a quad Core2 processor. They also found that despite an 8-fold theoretical performance gap between single and double precision, the GTX280 was nearly 2x faster than the same processor in DGEMM. The authors demystify the blocking mechanism and how they mapped the algorithm onto hardware. Unfortunately, when Nvidia introduced their Fermi architecture (e.g. GTX480 and Tesla c2050 cards), Volkov and Demmel's blocking routines no longer provided exceptional performance.

Quintana-Ortí et al used multiple GPUs to perform matrix multiplication and a Cholesky factorization[43]. They achieved 550 GFlops in single precision using 4 Tesla S870s by

adapting their FLAME runtime to work with multiple accelerators. This work represents some of the earlier attempts to use multiple GPUs to accelerate linear algebra problems.

MAGMA (Matrix Algebra on GPU and Multicore Architectures), among other things, improved on Volkov and Demmel's GPU kernels on Fermi hardware[73]. The novel feature in these kernels is an additional tiling onto registers in addition to shared memory. The SGEMM and DGEMM kernels found in MAGMA are to this day among the fastest on Nvidia hardware.

In [39], we implemented preliminary work on fast matrix multiplication using AMD's Radeon 5870. When running OpenCL-translated MAGMA GEMM kernels on the Radeon 5870, we found that while SGEMM performance virtually equaled the Tesla c2050 in absolute terms (at 600 GFlops) it lagged in terms of the card's peak performance. This led us to translate an IL (Intermediate Language, AMD's backend pseudo-assembly language) kernel into OpenCL.

The new OpenCL kernel, originally developed by Nakasato[74] in IL, featured radically different blocking mechanics than the kernels designed for Nvidia hardware. Each GPU work item tiled an 8x8 block of C into registers and streamed data through images. With these performance improvements, we found the Radeon 5870 could achieve 1.4 TFlops. The DGEMM version of our OpenCL kernel achieved 300 GFlops/s, virtually matching the c2050 in both absolute and achieved performance.

In having to write a variety of kernels, we arrived at three main conclusions. Firstly, CUDA and OpenCL usually feature nearly identical performance. Other work[75] agrees. Since they both feature nearly identical kernel languages, this is unsurprising. Secondly, OpenCL is less performant than low-level backend pseudo-assembly (IL). In our experiments, Nakasato's IL kernels outpaced our OpenCL kernels by 50%. Lastly, while OpenCL delivers portability in terms of correctness, it doesn't necessarily provide performance portability. This fundamentally arises from the facts that architectures are different and compilers aren't perfect. To overcome this last point, we propose a kernel generator with tuning parameters that can map to either Nvidia or AMD GPUs.

5.2 MAGMA on Tahiti

We started our work by examining the performance of our Nakasato-style OpenCL SGEMM and DGEMM kernels running on the Radeon 7970. Nakasato reports his IL implementation achieving 2.3 TFlops in SGEMM and 680 GFlops in DGEMM[76]. We found our OpenCL version of Nakasato's kernel achieved marginally better performance than on the Radeon 5870. The kernels originally designed for the Tesla c2050 however yielded a different story.

To examine the MAGMA kernel's blocking parameters in depth, we created a kernel generator Python script. This script accepts 4 parameters: TBR, TBC, TRR, and TCR. The first two parameters control the number of work items per row and column of each work group. The second two parameters determine the number of work items each work item computes in the row and column dimensions. Figure 5.1 shows the original kernel generation parameters we proposed in [39]. Our Python script assumes TR, TC, and VL are 1 and KB equals TBR. These additional values in principal allowed our kernel generator to create both the Nakasato and MAGMA style blockings (and hybridizations of the two) but remain as future work.

We used this kernel generator to explore the performance of different blocking techniques. We found that both DGEMM and SGEMM achieve their best performance when TBR and TBC are 16. This yields a 16x16 work group (256 work items), which can single handedly use an entire compute unit on Tahiti. We found that TRR and TCR equaling 6 yield the highest performance in both SGEMM and DGEMM. Under the 12.2 drivers, we found that setting these values to 4 gave best performance in DGEMM. Oddly enough, TBR=TCR=16 and TRR=TCR=6 are the optimal tuning parameters for the GTX 480 SGEMM as well. This suggests that performance is highly portable between the GTX 480 and Radeon 7970.

We benchmark both MAGMA's SGEMM and DGEMM kernels and compare against AMD's BLAS library. Our benchmark seeks to time only the matrix multiplication itself; data transfers, backend kernel compilation, and lazy allocations aren't included. To avoid lazy allocation overheads found in AMD's OpenCL library, our benchmark allocates matrices

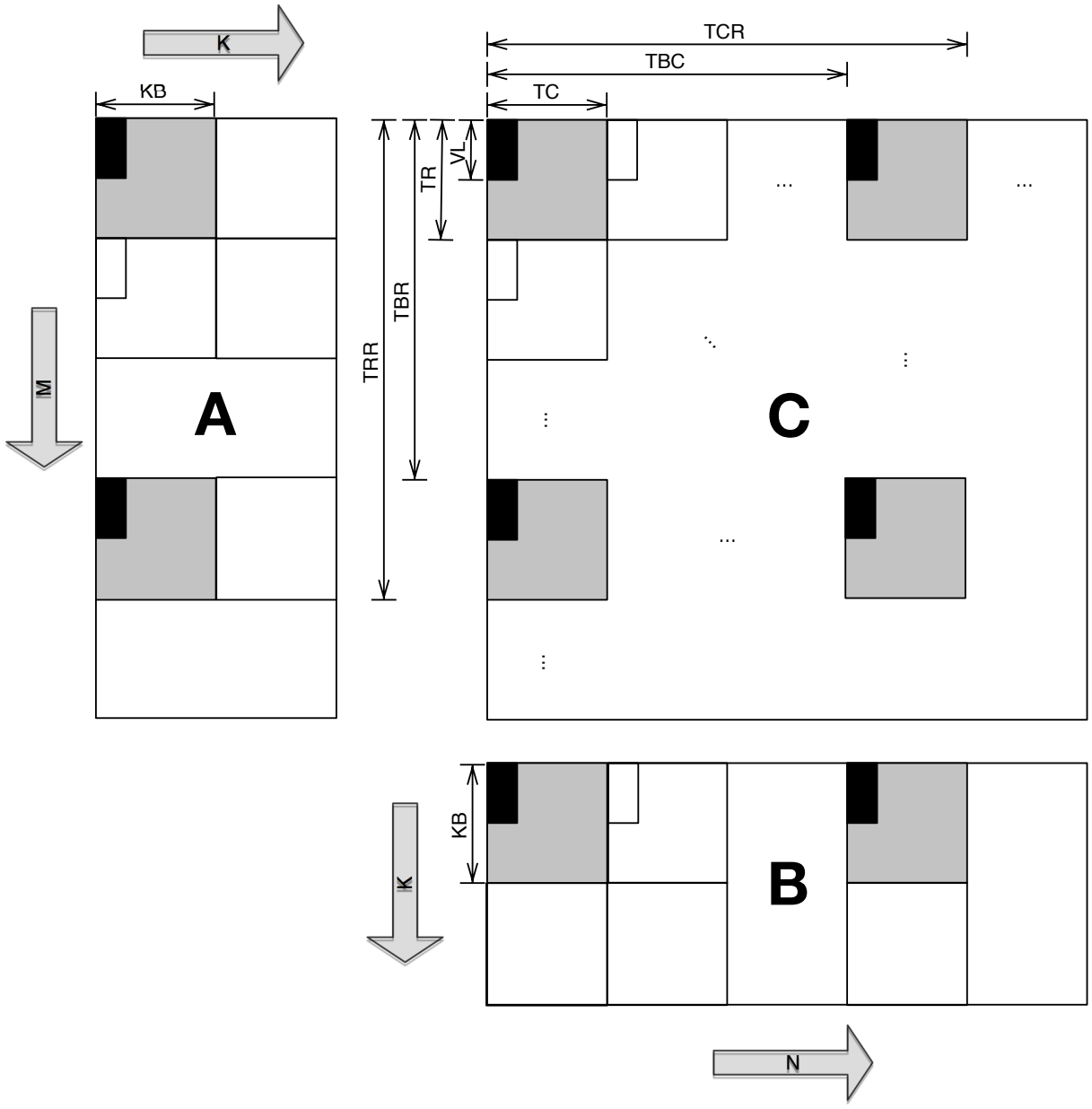

Figure 5.1: MAGMA Tuning parameters

A, B, and C on the host, transfers them to the GPU, enqueues a GEMM kernel, calls clFinish(), enqueues another GEMM on the same matrices, then calls another clFinish(). This ensures all lazy allocations occur on the first call to GEMM, allowing our timings of the second call to be accurate.

Figure 5.2 shows the performance of the MAGMA SGEMM and DGEMM kernels running on a single Radeon 7970. We acquired these results using the Catalyst 12.4 drivers. Under the 12.2 drivers, our kernels achieved peaks of 2 TFlops in single and 750 GFlops in double. However, the old drivers yielded scalability issues in our multi-GPU implementation related to data transfers. The newer 12.4 drivers eliminated these issues. As such, we present results using the more recent drivers for perspective in our multi-GPU work. We compare our results to AMDBLAS 1.7.257. MAGMA's SGEMM kernels yield nearly 300 GFlops over those found in AMD's BLAS library. MAGMA offers some improvement in DGEMM as well (90 GFlops).

5.3 Multi-GPU GEMM

My multi-GPU GEMM implementation leverages clUtil's ParallelFor() loop to parallelize over blocks of C. One problem with the vanilla GEMM algorithm is that it assumes column major storage, which makes transferring blocks of data to a device problematic. In principle, one could transfer a series of columns to copy blocks to and from devices. However, this would result in immense overheads as the driver must pin the column, copy a few kilobytes over the PCIe bus, and unpin the column. To overcome this limitation, we use Block Major Storage (BMS) for each matrix.

Under the BMS scheme, blocks of the matrix exist contiguously in memory. This allows our algorithm to copy an entire block with a single data transfer. In our implementation, blocks are ordinaled using columns as the leading dimension. Each block is itself a column-major matrix.

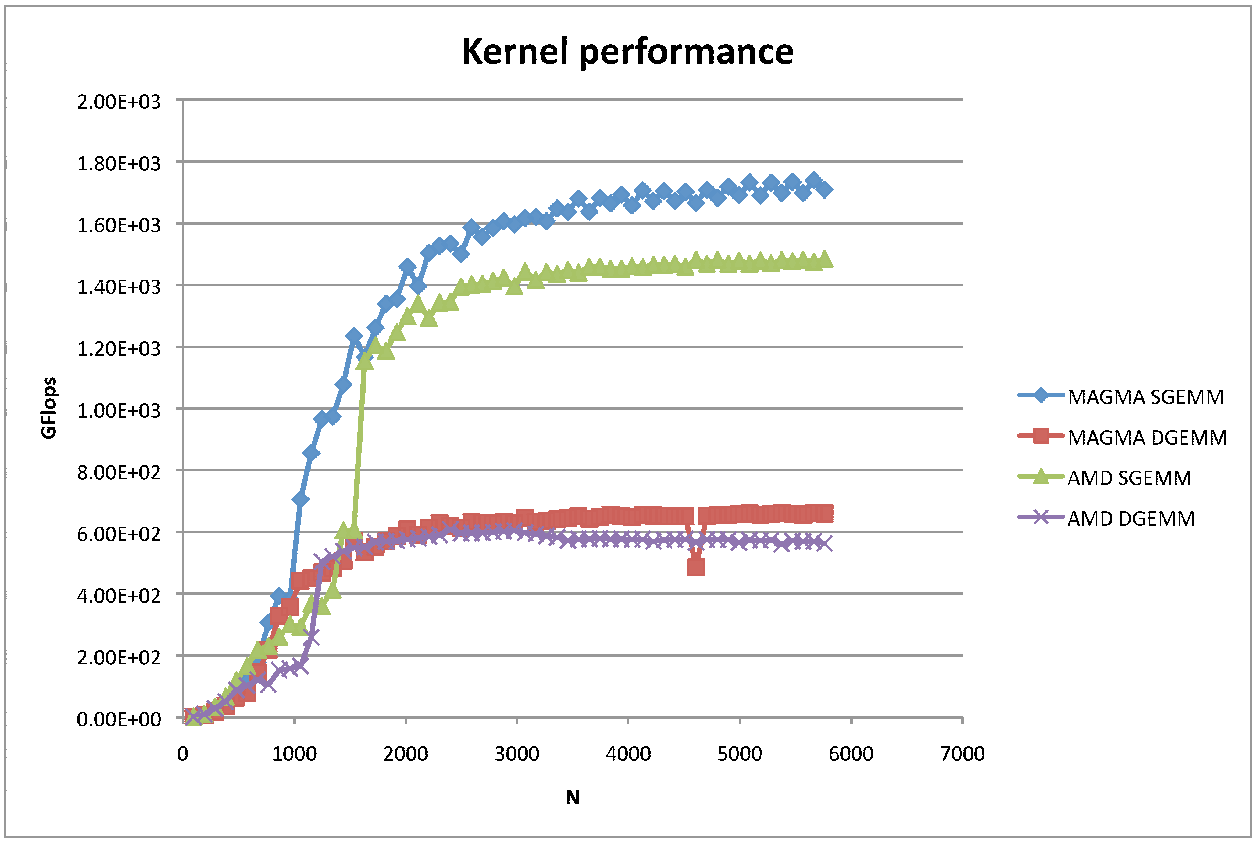

Figure 5.2: Magma SGEMM and DGEMM versus AMDBLAS 1.7.257 SGEMM and DGEMM

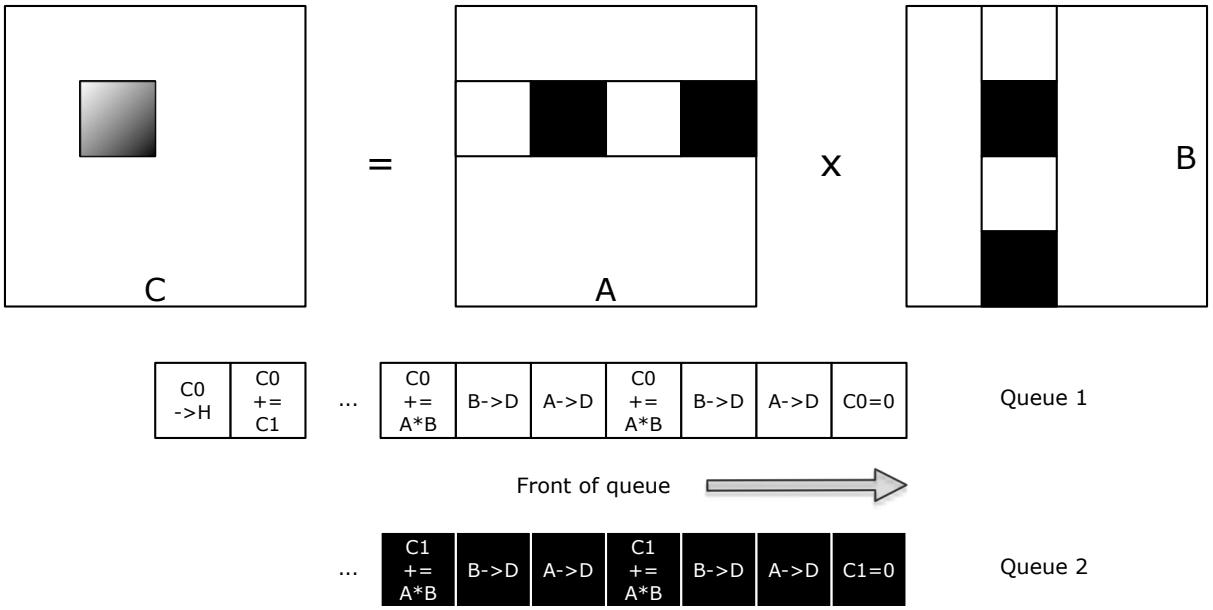

Figure 5.3: Algorithm for computing a given block of C

We preallocate and zero buffers for storing blocks of A, B, and C on each device. This allows us to eliminate lazy allocation overheads (which are very high on GPUs). We allocate two buffers for each on each device. This allows us to double buffer and use two command queues, which in principle should allow data transfers and computations to overlap.

Figure 5.3 gives the actual algorithm for computing a given block of C. First, the algorithm zeros the output C buffers. A serial for loop iterates over blocks in the k dimension of A and B. On each iteration, the algorithm enqueues transfers of the current block of A and B, and a GEMM with beta = 1.0. The algorithm then swaps buffers, switches the current command queue, and moves on to the next iteration. Once all blocks are enqueued, the algorithm then enqueues an accumulation of the two buffers and a copy of the final result back to the host. This ping-pong between the two queues allows one queue to be executing while the other queue is transferring A and B.

5.4 Scheduler performance

Matrix multiplication is an important application that allows testing for certain characteristics of scheduling algorithms. The algorithm itself is fairly simple to implement, in principle runs well even with static scheduling, is (usually) data agnostic with respect to performance, and often serves as the first litmus test for the viability of parallel computing methodologies[77]. Indeed, [52], [56], [57], [59] and [58] all explore their scheduling performance using matrix multiplication. In this respect, the work in this dissertation is no different; it compares the performance of heterogeneous scheduling in matrix multiplication.

One area where the work in this dissertation does differ is in the performance discrepancies between resources. The test machine used for GEMM experimentation features 3 Radeon 7970 GPUs and 32 Interlagos cores. The OpenCL GEMM kernel features performance given in 5.2 when run on a single Radeon 7970. Using a single GPU, the kernel peaks at 1.7 TFlops and 650 GFlops in single and double precision respectively. Thus, the combined throughput of all 3 GPUs exceeds 5TFlops in single precision and nearly 2 TFlops in double. This same kernel running on all 32 CPU cores treated as a single OpenCL device, however, yields a mere 30 GFlops in both single and double precision. This implies 166 and 66-fold performance discrepancies between the processors and GPUs. Using equation 2.1, one quickly sees a major scheduling challenge: in order to achieve perfect load balancing, a scheduler must give exactly 1 task to the CPU for every 166 tasks given to the GPU (66 in DGEMM). When combined with the large block size (5760) used in the presented multi-device GEMM algorithm, this implies that the CPU may provide extremely marginal benefit for problem sizes that no longer fit in the machine's generous 64GB of memory. Equation 2.1 predicts a 956,160 x 956,160 matrix is the smallest that could theoretically achieve perfect load balancing.

Another problem with using the CPUs in tandem is GPUs is that the processor schedules work on the processors while simultaneously running compute kernels. If a compute kernel delays scheduling of a GPU kernel or reduces memory bandwidth needed for data transfers, overall application performance can suffer.

A side effect of the algorithm we use (chosen to reduce data transfer overheads) is that with large block sizes, there exist few iterations to parallelize with, hurting opportunities to load balance. Unfortunately, the algorithm executes more efficiently with larger block sizes, as this reduces the amount of data transfers needed. In one extreme case, with a block size equaling the entire matrix, there exists no parallelism in our algorithm's ParallelFor loop, but the number of elements transferred equals $O(N^2)$. In the other extreme case, when the block size equals 1 element, there exist $O(N^2)$ parallel iterations, but the algorithm needs $O(^3)$ data transfers. Empirical testing shows that larger block sizes yield better performance scalability using 3 Radeon 7970s. An algorithmic improvement that should reduce block sizes while maintaining scalability is to use a cache as done in [43]. This remains as an exercise for the reader.

Timings in this section include all data transfers to and from the devices as well as loop overhead. We varied the matrix size while holding the block size constant at 5760. The machine tested has 32 Interlagos 6272 cores (2 sockets with 16 cores each), 3 Radeon 7970s (3GB), and 64 GB of PC1333 DDR3 RAM. The test machine's motherboard features 3 PCIe 2.1x16 slots. PCIe 3.0, which the Radeon 7970 supports, may yield even higher performance.

Figure 5.4 shows the performance of SGEMM and DGEMM running using all resources in the system. We ran this experiment using the algorithm in figure 5.3 using a static self scheduler with a constant chunk size of 1, the Enhanced Guided Self (EGS) scheduler (a reimplementation of [57]), and the PINA scheduler. Both the PINA and static scheduler execute with nearly identical performance (i.e. within each other's variance). This results from the fact that both schedulers assign work in exactly the same manner for different reasons. The static scheduler breaks each iteration into a chunk size of 1 by design. PINA used autotuning and found that a chunk size of 1 yields high performance on both the CPU and GPU. When CPU at N=5760 there exists only one task and it's given to a GPU. At N=11520, both schedulers assign one of the four tasks to each of the four devices. This amounts to the worst case load imbalance wherein all three GPUs execute their one job and wait on the OpenCL CPU device. As N increases, so does the number of tasks given to the

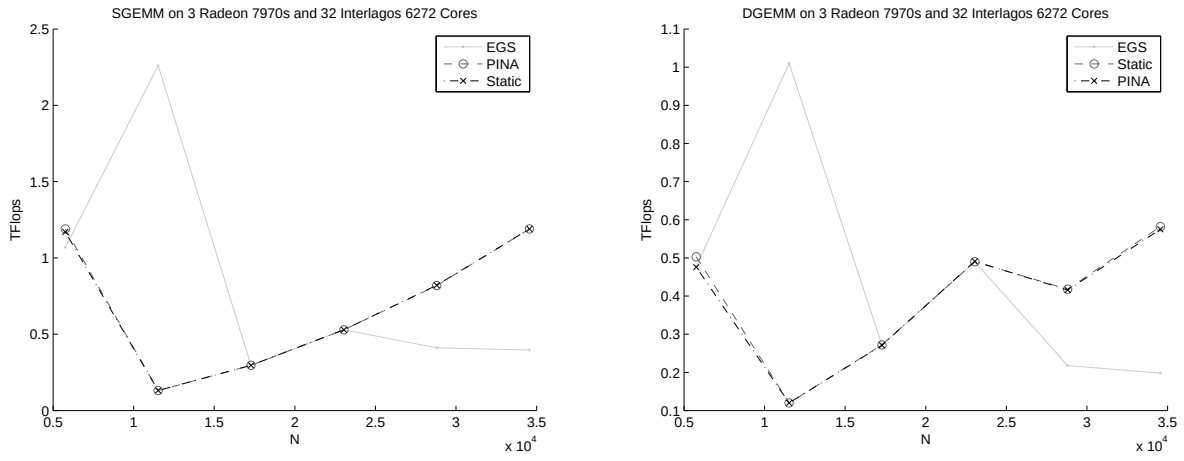

Figure 5.4: GEMM running on 3 Radeon 7970s and 32 Interlagos cores using static self scheduling, the EGS Scheduler, and the PINA scheduler

GPUs. Since the CPU is still busy, the GPUs take additional tasks. If this plot continued ad infinitum, a periodic sawtooth pattern would emerge with dips denoting problem sizes where the CPUs received an additional task.

The EGS scheduler tells a different performance story. With 1 task (N=5760), it effectively executes the entire GEMM on the GPU, as seen in the PINA and static schedulers. With four tasks (N=11520), however, performance increases markedly. This arises from how EGS computes chunk sizes, taking the remaining work and dividing by the number of device types. With four tasks and two device types (i.e. the Radeon 7970s and the Interlagos 6727s), the first chunk has 2 iterations and the next two contain a single iteration. This amounts to one GPU receiving two tasks and the remaining GPUs receiving 1 task each; the CPU receives nothing. When the number of tasks equals 9 and 16, the chunking algorithm gives only 1 task to the CPU and the rest to the GPUs. This results in similar performance to the other scheduling algorithms. However, as the number of tasks continues to increase, the chunking algorithm gives non-trivial amounts of work to the CPU, introducing performance degradation.

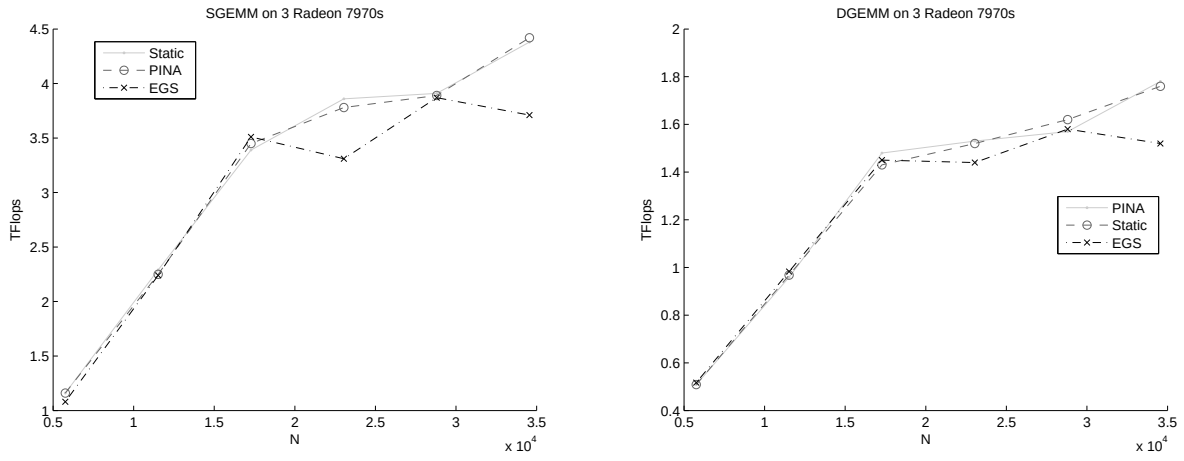

Figure 5.5: GEMM running on 3 Radeon 7970s using static self scheduling, the EGS Scheduler, and the PINA scheduler

Figure 5.5 shows the performance of the 3 schedulers running on only the 3 Radeon 7970s. Setting the CLUTIL_DEVICE_EMBARGO=3 environment disabled the CPU as an OpenCL device visible to clUtil. Under this homogeneous environment, each of the three schedulers yields markedly higher performance than the heterogeneous environment. Furthermore, since there exists only 1 device type, EGS actually reverts to the original guided self scheduling algorithm. For many problem sizes, all three algorithms feature nearly identical performance. However, in some cases, EGS performs notably worse than both static scheduling and EGS. This results from the non-unit chunk sizes allowed in EGS, which introduces load imbalance. The other two schedulers effectively perform pure self scheduling, which in principle allows for near optimal load balancing. This is because their chunk size is always 1.

The results in figure 5.5 also provide evidence that the actual scheduling overheads of all three algorithms is nearly identical. This arises from the fact that all three algorithms achieve nearly the same performance for most problem sizes. Aside from the EGS introduced load imbalance, the static and PINA schedulers achieve 83% of the theoretical throughput in both SGEMM and DGEMM. For larger problem sizes, EGS should do the same. Unfortunately, testing larger problems resulted in page faults and driver lockups.

The results presented in this section indicate that merely having a compute device doesn't mean that using it an algorithm is a wise venture. All three scheduling algorithms experienced catastrophic slowdowns when including the CPU as a compute device. A more performant kernel for the CPU could bring the performance between devices within an order of magnitude of one another. While static scheduling and EGS's pure runtime behaviors lack the information required to intelligently exclude devices from computation, PINA's autotuning provides performance information that could easily allow it to disable devices as a function of asymmetry and number of iterations. [59] disables devices based on asymmetry with their predictive algorithms.

Chapter 6

Specmaster: A fast peptide search algorithm using OpenCL*

In shotgun proteomics, there exist two primary methods for identifying proteolytic peptides in mass spectrometry data: de novo and database searches. The former is useful for identifying novel proteins in samples, and is beyond the scope of this paper. Database searches require prior knowledge of what proteins could exist in the sample obtained through some other method, such as sequencing genome sequencing. A number of competing tools, such as X!Tandem[78], Sequest[79], Myrimatch[80], and Mascot[81] all perform database searches using a .fasta database on mass spectrometry data converted into one of a variety of formats (.dta, .mzXML, etc).

Initially, database search engines operated on single-core processors and weren't suitable for large complex searches due to search times. However, many modern database search algorithms use multi-threading to take advantage of multi-core processors, including Sequest, Myrimatch, and X!Tandem. Alongside other improvements in processor design and increased memory, current database search programs are markedly faster than their previous versions

*This section features excerpts from "For Three Easy Payments: Scoring Peptides With Portable Performance Using Specmaster," a conference paper I submitted to SAAHPC. I am responsible for its contents and the underlying research. The co-authors served as PI and advisory roles.

because they exploit multi-core CPUs. However, as most applications needed reprogramming to reap the benefits of multi-core processors, another redesign wave approaches (arguably, has already arrived) with the growing ubiquity of heterogeneous computing.

Few database search engines can use other accelerators such as GPUs. The most notable GPU-accelerated database search algorithm is FastPaSS[82], but this requires specialized peptide databases. This makes FastPaSS unsuitable in pipelines that use more traditional .fasta databases. Another project, gMacro[83], seeks to accelerate Sequest using Nvidia GPUs. Both projects use CUDA[29], limiting portability, and both compare relative performance against a single core. In this paper, we present Specmaster, an OpenCL database search algorithm.

Specmaster overcomes some of the limitations of previous work. Being based on Myrimatch, Specmaster uses standard .fasta databases and .mzXML spectra files while providing portability to multi-core processors, both Nvidia and AMD GPUs, and future devices that will support OpenCL such as Altera FPGAs[38]. We compare Specmaster running on an Nvidia GTX 480, an AMD Radeon 7970, two AMD Interlagos processors, and two Intel Sandy Bridge CPUs against Myrimatch running with 32 threads on the AMD and Intel processors.

6.1 Specmaster Algorithm

Specmaster implements the Multi-Variate Hypergeometric (MVH) scoring algorithm found in the original Myrimatch paper[80]. Given a set of peptides generated from a .fasta protein database, Specmaster scores MS2 or higher scans using an MVH distribution. MS2 and higher scans contain fragmented peptides as opposed to MS scans, which contain intact peptides. This fragmentation process creates a signature usable for identifying peptides; MS scans don't contain this information and thus aren't scored. Figure 6.1 gives a high-level description of Specmaster's algorithm using pseudo-C. In essence, the program executes in 4 steps until it scores all peptides found in all proteins.

```
proteinsRemaining = numProteins;
peptideBuffer[kBufferSize];
scores[numScans][5];

while(proteinsRemaining > 0)
{
  proteinsRemaining -=
    fillBuffer(peptideBuffer);

  for(each scan in spectrum)
  {
    preprocess(scan);
    findCandidates(scan, peptideBuffer);

    for(each peptide in candidates)
    {
      curScore = score(peptide);
      insertScore(curScore,
                  scores[scan]);
    }
  }
}

for(each scan in spectrum)
{
  writeScores(scores[scan]);
}
```

Figure 6.1: Specmaster's algorithm

6.1.1 Generating peptides

Because memory on accelerators is usually limited, they often can't hold all peptides derived from an entire fasta file at any given time. To overcome this (and to conserve host memory), Specmaster processes peptides in groups that do fit in memory (currently a compile-time constant of 1GB). It loops through the list of proteins creating peptides according to experimental rules, appending them to a peptide buffer. If the additional peptides derived from the current protein would overflow the peptide buffer, Specmaster discards the peptides and remembers to start at this protein for the next batch. This step embodies the fillBuffer() call in figure 6.1. The rules for generating peptides vary depending on experimental procedures.

Upon reading a protein from the user-defined fasta file, Specmaster creates peptides using a variety of rules. Specmaster currently supports tryptic digestion, which creates cleavages after lysine (K) or arginine (R) not followed by a proline (P). In an ideal world, trypsin would always break peptides at every cleavage site. However, in practice, missed cleavages are quite common. As such, specmaster generates peptides assuming 0,1,...k missed cleavages, where k is a user defined parameter. By default, Specmaster assumes k=UINT_MAX. If the generated peptide is greater than 63 amino acids, Specmaster discards it because extremely long peptides rarely (if ever) appear in MS data, and this artificial limit allows Specmaster to efficiently store peptides. Specmaster packs amino acids into 64 bytes: the first containing the length of the peptide and the remaining 63 containing the amino acid sequence. Figure 6.2 gives an example of how Specmaster creates peptides from proteins.

Specmaster checks if a generated peptide already exists in the buffer by searching for the sequence in an C++ STL multimap (e.g. a red-black tree). If the sequence is unique, Specmaster appends it to the peptide buffer. If found, it adds a reference to the protein for the sequence into the multimap and doesn't add the sequence to the peptide buffer. This multimap often becomes very large (several to 10s of GB). Optimizing the time spent

Protein:
MK LYNLK DHNER QVSFK A
Peptides:
MK
LYNLK
DHNER
QVSFK
A
MK LYNLK
LYNLK DHNER
DHNER QVSFK
QVSFK A
MK LYNLK DHNER
LYNLK DHNER QVSFK
DHNER QVSFK A

Figure 6.2: Peptide generation example using tryptic digestion with up to 2 missed cleavages

performing redundant computation versus time and space spent avoiding it remains as future work.

Specmaster computes a peptide's neutral mass when appending it to the peptide buffer. A peptide's neutral mass is simply the sum of its amino acids.

In addition to generating peptides from the proteins themselves, Specmaster generates peptides from a distractor set used to identify false positives. The distractor set consists of the reverse of every protein in the user given fasta file. Using the distractor set, other tools in the proteomics pipeline can compute the score threshold that yields positive identification with a given false discovery rate.

When the peptide buffer fills, Specmaster sorts the peptides based on their mass. This vastly simplifies candidate generation after scan preprocessing.

6.1.2 Scan Preprocessing

With a batch of digested peptides, Specmaster transfers the batch of peptides, their masses, a batch of scan headers, and their data to the selected OpenCL device. Since the peptide buffer is easily the largest data structure on the device and is by default 1GB, Specmaster packs peptides into a virtual 1D image with 32-bit unsigned RGBA channels. Each peptide consists of 4 pixels in the image (4 bytes/channel * 4 channels/pixel * 4 pixels = 64 bytes). clUtil[†] provides macros for mapping 1D image coordinates onto a 2D image, since OpenCL 1.1 doesn't allow 1D images. Using images instead of buffers overcomes allocation limitations on some AMD GPUs (namely, the Radeon 5000 series). In the future, we plan to use a parallel for loop to score multiple scan batches concurrently using all OpenCL devices in a machine. Specmaster then preprocesses scans on the selected OpenCL device.

Not all scans in mass spectrometer datasets are of sufficient quality to identify peptides. Level 1 scans contain a variety of intact peptide molecular masses alongside possible impurities. Since these scans don't contain fragmented peptides, Specmaster skips level 1 scans. Level 2 (and possibly higher, experiment dependent) scans do contain peptide fragments (or fragments of fragments in higher level scans). For level 2 scans and higher, Specmaster computes the Total Ion Current (TIC i.e. the sum of all peak intensities for a given scan). It then sorts the peaks by intensity and removes all peaks after a fractional threshold of the cumulative TIC. By default, this threshold is 98% of the TIC, but this is user configurable. If the scan still contains more than some user definable number of peaks (301 by default), Specmaster keeps only the most intense peaks up to this number. Specmaster also removes peaks down to a multiple of $2^c - 1$. If the scan has fewer than $2^c - 1$ peaks, Specmaster marks the scan as bad and doesn't search it. In this case, c refers to the number of intensity classes, a user defined parameter used in scoring.

After removing peaks, Specmaster replaces the peak intensity with a class number. Myrimatch's (and thus Specmaster's) scoring algorithm requires associating peaks with one

[†]clUtil is a library we designed to dramatically improve programmer productivity in OpenCL

of c intensity classes. Class k is twice as large as class $k-1$ and the sum membership of all classes equals the number of filtered peaks. Thus, in a scan with 49 filtered peaks remaining, the 7 most intense are class 1, the next 14 are class 2, and the bottom 28 are class 3 when $c = 3$. Specmaster overwrites the peak's intensity with its class number. Specmaster additionally computes other metrics needed in scoring.

After removing small peaks, Specmaster computes other metrics needed for scoring and resorts the data by mass to charge to simplify the scoring algorithm.

Specmaster preprocesses scans in parallel by assigning different work groups to different scans while collaboratively using a device specific number of work items per group to perform the preprocessing tasks required for a given scan. This amounts to launching the preprocessing kernel with $numScans$ x k work items with 1 x k work items per work group. $k = 64$ on graphics accelerators and $k = 1$ for CPUs. We designed the kernel to run correctly regardless of k.

Figure 6.3 gives an example of a k-agnostic loop that computes the total ion current of a scan. In this case, we define agnostic to mean that the kernel neither knows nor cares at compile time how many work items exist in a work group. In this example, Specmaster defines kItemsPerWorkGroup in a header file according to the number of items per work group it determines optimal for the given device. In addition to working regardless of the number of work items per group, this kernel performs coalescing and avoids bank conflicts on GPUs.

6.1.3 Packing

Specmaster supports reading spectra using one of three methods. It can pack spectra into images, load them from global memory, or read them from shared memory. Specmaster uses preprocessor directives combined with branches in host code to correctly implement the desired method. Each of these methods provides different performance characteristics on different devices. Specmaster can change the method used on a device-by-device basis,

```
__local uint TIC = 0;
__local uint tmp[kItemsPerWorkGroup];

for(unsigned int i = get_local_id(1);
    i < numPeaks;
    i += get_local_size(1))
{
  tmp[get_local_id(1)] +=
    intensity[i];
}

barrier(CLK_LOCAL_MEM_FENCE);

if(get_local_id(0) == 0)
{
  for(unsigned int i = 0;
      i < get_local_size(1);
      i++)
  {
    TIC += tmp[i];
  }
}

barrier(CLK_LOCAL_MEM_FENCE);
```

Figure 6.3: Local work size agnostic code fragment to compute total ion current

providing portable performance on a wide range of accelerators and processors. When loading from images, Specmaster packs the spectrum data into two $kMaxPeakCount$ x $numScans$ RGBA images. Only the red channel contains spectrum data and the remaining channels are unused. This ensures high portability, since RGBA is a required format in OpenCL implementations that support images. Using the other two methods, OpenCL packs the mass to charge ratio (m/Z) and peak classes into two $numScans$ x $kMaxPeakCount$ array. After packing the spectrum, Specmaster scores the scans.

6.1.4 Finding Candidates

Specmaster finds and scores candidates both in the same kernel. If the mzXML provides precursor charge state information, Specmaster assumes the provided charge. If not, it assumes charge states 1, 2, ... z, where z is a user configurable parameter for the maximum assumed charge state. For each charge state, Specmaster first finds candidate peptides that need scoring and then scores them.

To find candidates, Specmaster computes the neutral mass for the precursor m/Z with the currently assumed charge state (equation 6.1). Since the peptides are sorted by mass, the candidate selection function performs two modified binary searches: one finds the index of the least massive peptide within the user defined precursor tolerance of the neutral mass while the other search finds the index of the most massive peptide in the tolerance. By default, the precursor tolerance is ±1.5 Da/Z, but users can change this to any value. Only work item 0 in the work group performs this search due to the difficulty (and marginal utility) of implementing an efficient parallel binary search on up to 16 million masses.

$$m_{neutral} = \left(\frac{m_{precursor}}{Z_{precursor}} - m_{proton} \right) Z_{assumed} \tag{6.1}$$

Table 6.1: Possible fragmentation patters for the "PEPTIDE" amino acid sequence

B-ions	Y-ions
	PEPTIDE
P	EPTIDE
PE	PTIDE
PEP	TIDE
PEPT	IDE
PEPTI	DE
PEPTID	E
PEPTIDE	

6.1.5 Scoring

With start and end indices found for candidate peptides, Specmaster now scores all candidates in this index range. Each work item in a group scores a different candidate for the same scan. This allows significant reuse of peak data through either texture caches, local memory, or general purpose data caches, depending on the device configuration; work items continually query the same set of peaks until all candidates for all assumed charge states are exhausted. The loop over candidates executes in parallel in a manner agnostic to work-group size. This allows the same kernel to function correctly regardless of local work size.

To score a candidate peptide, the kernel searches for B and Y ions for every amino acid in every charge state 1, ... z, where z is the assumed precursor charge state. Tables 6.1 and 6.2 list the possible fragmentation patterns and charge distributions for an example peptide. Since the mass spectrometer pulls in many ions and fragments them simultaneously, a scan could theoretically contain peaks for each possible B and Y ion for each charge state. In practice, scans almost always have missing peaks for a variety of experimental reasons (e.g. signal integrity, insufficient precursor ions, non-uniform fracturing probabilities, and non-uniform proton distributions). As such, scoring algorithms should handle missing peaks.

To search for a peak associated with a given charge state and amino acid sequence, Specmaster performs a modified binary search that minimizes error between the calculated

Table 6.2: Possible proton distributions between B and Y ions for +3 charged precursor ion "PEPTIDE"

B-ions	Y-ions
PEP+3	TIDE+0
PEP+2	TIDE+1
PEP+1	TIDE+2
PEP+0	TIDE+3

m/Z for that peak and the actual peaks in the scan data. If the error is within ±kPeakTolerance, the algorithm found a peak for that fragment. The default value for kPeakTolerance is 0.5Da, but is user configurable and optionally specified in parts per million. Specmaster keeps track of how many peaks it finds in each intensity class.

Equation 6.2 describes how to compute the probability of the peptide appearing by chance using a multivariate hypergeometric distribution. Equation 6.3 yields the actual score that Specmaster reports. When computing the MVH, m_i denotes the total number of peaks in that intensity class and t_i denotes the number of peaks found in the spectrum. The $i = 0$ case is the "void" class; contains the number of places a peak could exist within the peak tolerance range but doesn't and t_0 represents the number of peaks in the scan range not found. T and M are the total number of places peaks could exist and be outside of each other's tolerance in the scan range and M represents the total number of peaks searched in the scan range.

$$p = \frac{\prod_{i=0}^{c} \binom{t_i}{m_i}}{\binom{T}{M}} \tag{6.2}$$

$$s = -\ln(p) \tag{6.3}$$

Directly computing m choose k is problematic for even modest values of m and k. To overcome this, we compute the natural log of m choose k. We precompute natural logarithms of the first 8192 factorials in 32-bit floating point.

Each work item maintains a top k list of results, k defaulting to 5 and being configurable. When a work item computes a peptide score, it inserts it into its local top k array and marks the index of the corresponding peptide. When the kernel scores all peptides individually the work item with local id 0 coalesces the top k lists of all work items and then the kernel exits. Finally, Specmaster transfers the scores and peptide indices back to the host and coalesces the new peptides and their scores with a master top-k list of all peptide batches.

Like the preprocessing and pack kernels, the candidate / scoring kernel is work size agnostic. Furthermore, this kernel uses the reqd_work_group_size() attribute in OpenCL to allow the compiler to optimize away local memory barriers in some cases. Specmaster launches this kernel with a $numScans$ x k global work size with a 1 x k local work size. Again, k can vary depending on the OpenCL device Specmaster uses for scoring. The first dimension of the work grid parallelizes over scans, while the second dimension parallelizes over candidate peptides, allowing the kernel to run efficiently on GPUs' SIMD multiprocessors.

6.1.6 Data dependent performance

The time taken to process a scan is a function of the number of candidates Specmaster has to evaluate. If a scan's precursor mass to charge ratio (M/Z) isn't indicative of a peptide (e.g. the scan is junk) or the M/Z doesn't correspond to many candidates, then Specmaster should process that scan quickly. However, if the scan's M/Z happens to correspond with a large number of peptides derived from the database, then Specmaster has to score a large number of peptides and processing that scan takes a longer amount of time. Since Specmaster processes multiple scans concurrently in a single kernel call, the total time to process a batch of scans is also a function of the number of processors in the device processing them and how the OpenCL runtime schedules scans on them.

Figure 6.4 shows the time taken on three different datasets. Each dataset is 12 different files totaling over 100,000 scans and has a corresponding protein database. As these graphs indicate, each dataset has a unique "shape" to its execution time as a function of the

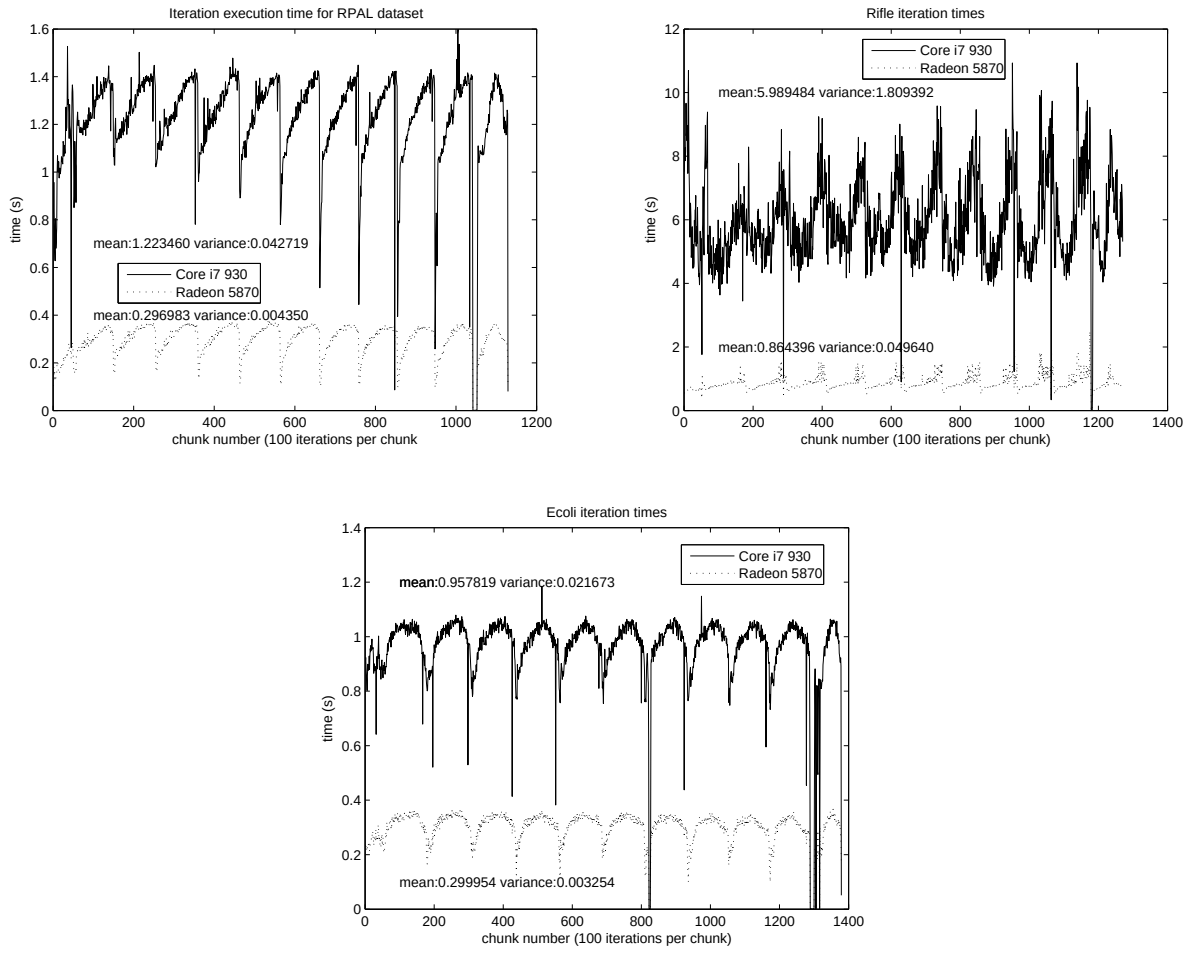

Figure 6.4: Execution time vs. batch number on Core i7 920 CPU and Radeon 5870 GPU on 3 different datasets

iteration. Furthermore, Figure 6.5 (derived by dividing the top curve by the bottom of each respective plot) shows that the speedup of using a GPU changes as a function of both iteration and dataset. Both of these issues yield considerable difficulty in load balancing with static compile-time techniques.

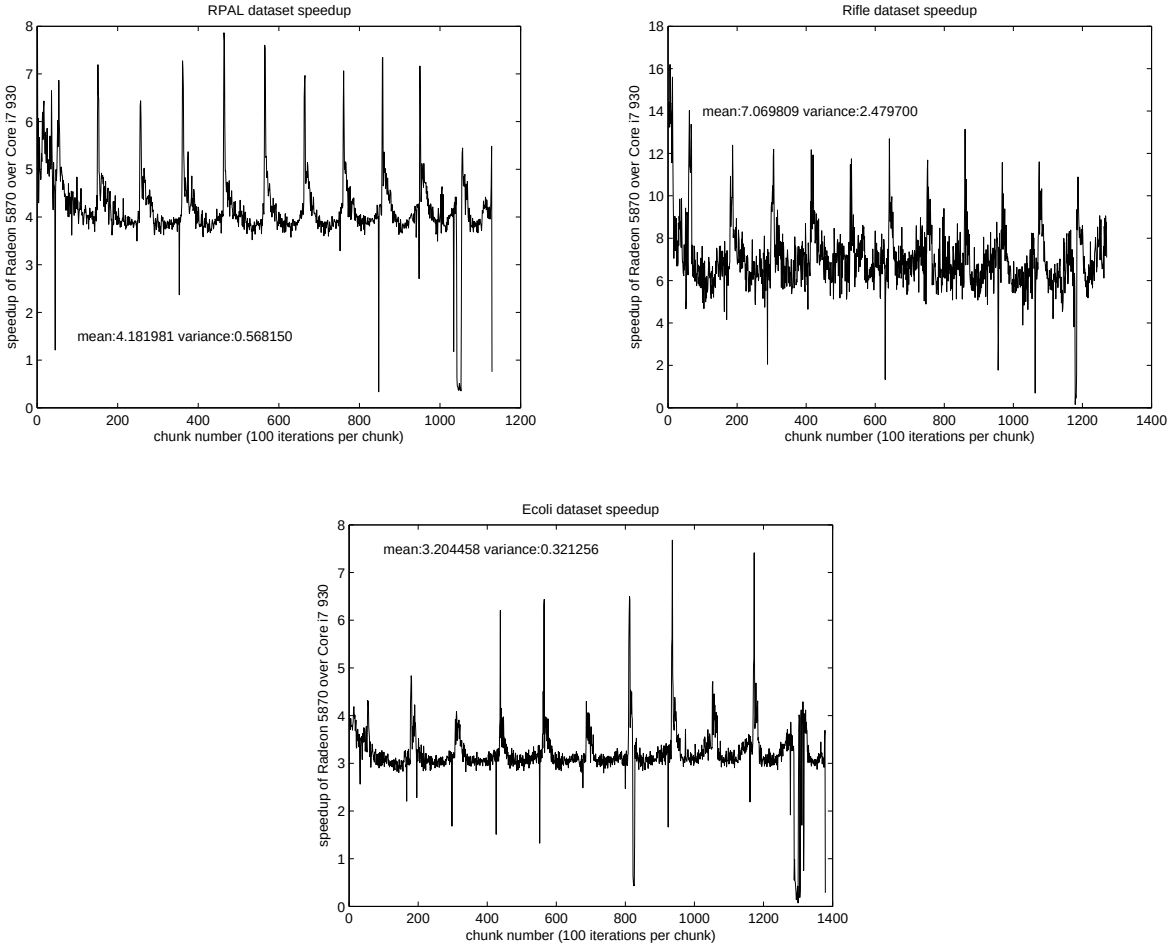

Figure 6.5: Speedup of Radeon 5870 over Core i7 920 on 3 different datasets

6.2 Portable performance

To achieve portable performance, we use two main programming tricks. Firstly, we design all kernels to provide two levels of parallelism and run in a work group size agnostic manner. Secondly, we abuse OpenCL's preprocessor to select kernel features on a device-by-device basis.

Our previous example in figure 6.3 gives an example of developing a workgroup-agnostic kernel. The kernel runs correctly with 1, 2, 64, or (theoretically) 10,000 work items in dimension 1. In practice, resource allocation will provide some upper limit on the size of work dimensions, but this limit should hopefully far exceed that needed for high performance. We use knowledge of how OpenCL implementations map kernels on specific devices to choose the appropriate work size.

We found that AMD's x86/x64 OpenCL implementation schedules different work items in a group to the same POSIX thread. Different work groups, however, can execute on different threads. Since the costs associated with switching work item contexts is non-trivial, running kernels with only 1 work item per group yields highest performance in this implementation. Intel's OpenCL x86/x64 implementation can potentially schedule multiple work items into different SIMD lanes. Using more work items in this case remains for future study.

With both AMD and Nvidia's GPU OpenCL drivers, we find that 64 work items per group provides good performance. On AMD's high-end GPUs, 64 work items equals one wavefront and on Nvidia's GPUs, 64 work items is a multiple of the warp size. This may change in future architectures, but Specmaster's flexibility allows us to trivially change the work size for these devices.

6.2.1 Exploiting the Preprocessor

We use OpenCL's runtime compilation and abuse its preprocessor to dynamically change Specmaster's behavior at runtime. Specmaster achieves this by writing a header file included

```
#define kTICCutoffFactor 0.980000f
#define kMaxPeakCount 301u
#define kPrecursorTolerance 1500u
#define kFragmentMZTolerance 0.500000f
#if CLUTIL_DEVICE_ID == 0
#define IMAGE_SPECTRUM
#define kNumThreadsPerBlock 64u
#endif
#if CLUTIL_DEVICE_ID == 1
#define GLOBAL_MEMORY_SPECTRUM
#define kNumThreadsPerBlock 1u
#endif
```

Figure 6.6: Example header created by Specmaster dictating kernel compilation

by kernels at runtime just before compilation containing a list of constants and preprocessor directives. Figure 6.6 gives a truncated example of such a header. Using this method, variables in an off-line compiled program become constants in an online compiled program; since many user defined parameters such as the TIC cutoff and max peak count never change once the program runs, runtime compilation can exploit this by defining them as constants in a header created at runtime.

By replacing variables with "constants," kernels can have fewer parameters resulting in cleaner code. Furthermore, this also provide opportunity for the compiler to better optimize the kernel. For example, when a loop trip count becomes known at compile time, developers can precede it with *#pragma unroll kTripCount* to fully unroll it at runtime in some OpenCL implementations. Another optimization compilers can now do is constant collapsing; most compilers can replace the expression a+21*3 with a+63.

One prominent example of this is using either shared, global memory, or images for storing spectra. Figure 6.7 shows how Specmaster can select which memory to use. Different memories have varying bandwidth and latency characteristics on different devices. For example, CPUs don't have hardware image sampling hardware, imposing additional software overhead for loading from them on the CPU. Furthermore, *__local* buffers map to main

memory like any other buffer, so there is no benefit to using them over *__global* memory. As such, on the CPU, Specmaster defines *GLOBAL_MEMORY_SPECTRUM*.

GPUs, on the other hand, have special hardware for loading from images and these stream through a cache hierarchy on virtually every GPU in existence. As such, they are often a viable mechanism for exploiting data reuse. Local memory may provide varying bandwidth depending on memory access patterns, device, and the application. Global memory may stream through a cache hierarchy; if not, using global memory is an inefficient mechanism to reuse data.

On the GTX 480, there is virtually no performance difference between using global and local memory. We found the Nvidia OpenCL runtime can't allocate a 1GB image, making them less desirable for storage. On the Radeon 7970, global memory yields marginally better performance than images and significantly better performance than local memory. This behavior likely results from Specmaster's wildly divergent memory access patterns when doing binary searches on the spectrum using multiple threads; bank conflicts abound. The Radeon 7970 features a full cache hierarchy for global memory. On the Radeon 5870, one must use images to allocate large buffers. Using preprocessor tricks, Specmaster can support all of these devices, even in the same machine.

One final way we use preprocessor exploitation is to remove branches. For example, Spemaster can measure peak tolerances in absolute Daltons or parts per million. Since Specmaster never changes its method mid-run, it can use preprocessor macros to do one or the other.

6.3 Differences with Myrimatch

Specmaster trades some flexibility for performance when compared to Myrimatch. For example, Specmaster requires that each intensity classes must contain twice as many peaks as the previous, whereas Myrimatch allows users to configure this.

```
bool find(
#if defined(GLOBAL_MEMORY_SPECTRUM)
          __global uint* mz,
          __global uint* peaks,
#elif defined(LOCAL_MEMORY_SPECTRUM)
          __local uint* mz,
          __local uint* peaks,
#else
          read_only image2d_t mz,
          read_only image2d_t peaks,
#endif
          uint numPeaks,
          uint searchMZ,
          uint* peakIntensity)
{
  int minIndex = 0;
  int maxIndex = numPeaks - 1;
  unsigned int minErr = UINT_MAX;
  unsigned int minErrorIdx = 0;

  do
  {
    unsigned int midPoint =
      (minIndex + maxIndex) / 2;

#if defined(LOCAL_MEMORY_SPECTRUM) ||\
    defined(GLOBAL_MEMORY_SPECTRUM)
    unsigned int curMZ = mz[midPoint];
#else
    int2 coord = { midPoint,
                   get_global_id(1) };
    uint4 curPixel =
      read_imageui(mz, s0, coord);
    unsigned int curMZ = curPixel.x;
#endif
    //...
  }while(minIndex <= maxIndex &&
         minErr != 0);
  //...
}
```

Figure 6.7: Using OpenCL's C preprocessor to change which memory Specmaster uses as a function of the device

Specmaster converts all input m/Z peaks to milliDaltons and performs most arithmetic using 32-bit unsigned integers. In principal, we could convert all mass values to 1e-5 Da, giving extra precision while maintaining a dynamic range up to 42,949.67295 Da. Using integers yields more precision than 32-bit floating point without the extra bandwidth (and hardware) requirements of double precision. Myrimatch uses double precision for all peak calculations. This can result in different roundoff characteristics between the two implementations.

Specmaster drops peaks in filtering to a multiple of the sum of class sizes. This exactly distributes peaks into classes, allowing easy on-the-fly calculation of class sizes.

In addition to actual implementation variances, Myrimatch can perform additional peak filtering and centroiding, charge state modeling, and supports a variety of other features not presently found in Specmaster, such as dynamic post-translational modifications[80], a feature we hope to implement in the future to leverage Specmaster's high throughput.

The culmination of these discrepancies results in different scores between Myrimatch and Specmaster. For low scoring peptides, this may result in different answers. However, high-scoring peptides themselves are the same in both programs.

Because Specmaster has different methods for coalescing scores depending on the number of work items per block, entries with the same score may permute between runs on different devices. Because of this, when we add device parallelism to Specmaster, results may actually vary from run to run. However, Specmaster still does guarantee that each score in the top 5 list is greater than or equal to the next and ensures a total ordering of all candidates outside of ties.

6.4 Single device performance

6.4.1 Experimental Setup

We use a variety of machines and setups to benchmark Specmaster using 4 different OpenCL devices. Table 6.3 lists the tested machines. We time Myrimatch running on both Intel Sandy Bridge processors and AMD Interlagos processors with 32 threads. The latter forms our comparison baseline. We then time Specmaster running on both processors, the Radeon 7970, and the GTX 480. All results presented time the entirety of program execution.

Table 6.3: Machines tested

	Machine 1	Machine 2
Processor	2x AMD Interlagos 6272	2x Intel Sandy Bridge E5-2680
Number of cores / threads	32/32	16/32
Base frequency	2.1GHz	2.7GHz
Memory	64GB	64GB
Accelerators	3x AMD Radeon 7970	None
	Machine3	
Processor	2x Intel Nehalem X5570	
Number of cores / threads	8/16	
Base frequency	2.93GHz	
Memory	24GB	
Accelerators	1x Nvidia GTX 480	

We test Specmaster on three datasets: rpal, rifle, and amd. Each dataset roughly contains the same number of scans, but have different number of proteins. The rpal dataset contains the isolate microbe R. Palustris and yields 1.3 million peptides to search. The rifle dataset contains a ground-water microbial community obtained in Rifle, CO; this dataset yields 6.8 million peptides. Finally, the amd dataset represents our largest search, generating 9 million peptides, contains a low complexity microbial community from an acid mine in California. We perform fully tryptic digestion on all datasets with unlimited missed cleavages.

We use the AMD APP SDK version 2.6 for both CPUs and the Radeon 7970. With the AMD GPU, we use the 12.4 Catalyst drivers. We tried testing the Intel OpenCL library as

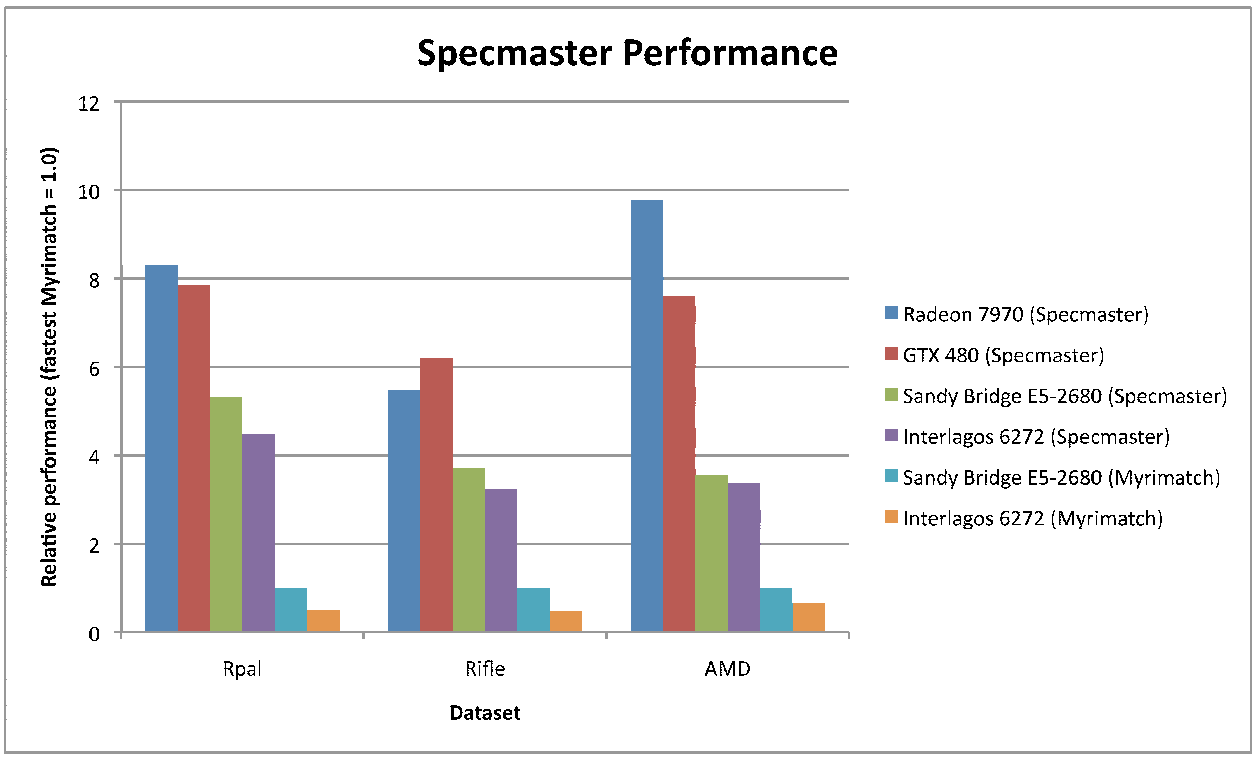

Figure 6.8: Specmaster end-to-end relative performance versus Myrimatch running on a 32 core E5-2680

well, but encountered segmentation faults related to libnuma. For the GTX 480, we used the CUDA SDK with OpenCL 1.1 and driver version 285.05.32.

6.4.2 Results

Figure 6.8 shows the relative execution rates of the same search run in both Myrimatch and Specmaster on a variety of hardware. In all cases, we compare Specmaster to Myrimatch using 32 CPU threads (i.e. Myrimatch is *not* a naive single-threaded implementation). When using the Radeon 7970 and GTX 480, we use a single GPU exclusively when identifying peptides.

Figure 6.9 shows the relative performance the peptide search portion of Specmaster running on each device. The other main component of Specmaster, the peptide generation,

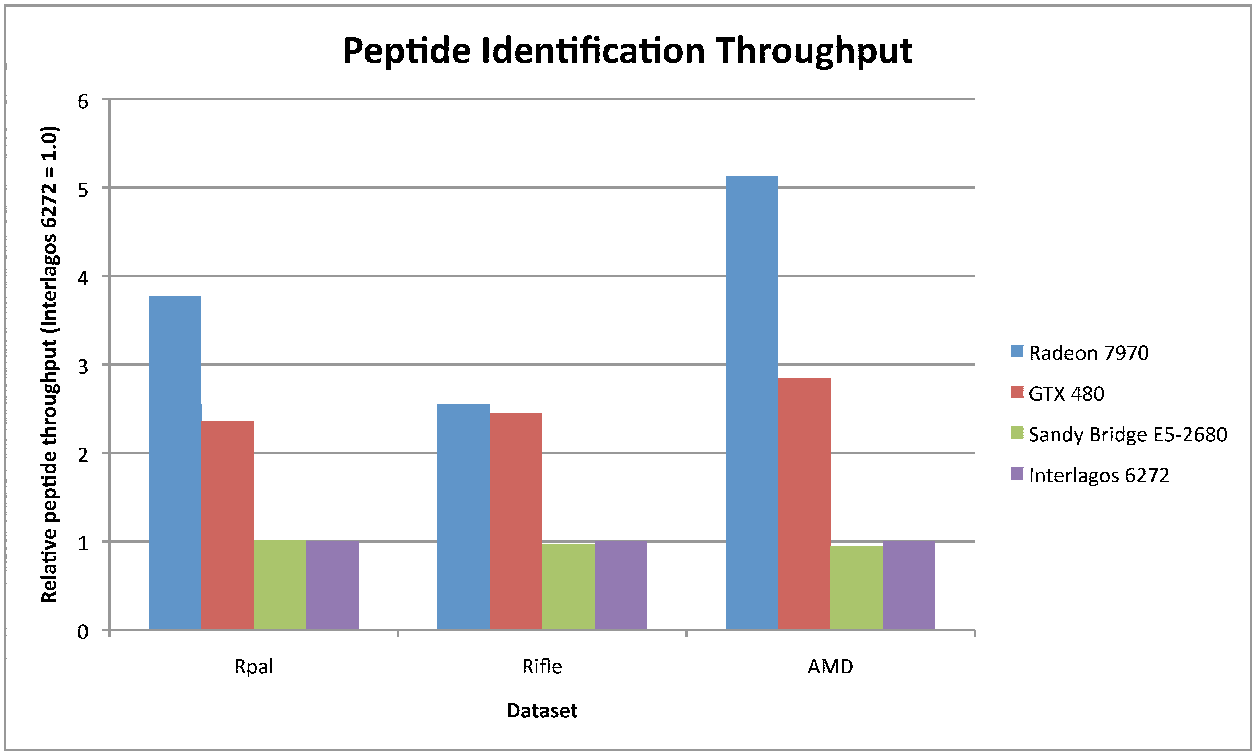

Figure 6.9: Relative throughput in parallelized peptide identifications section of specmaster

runs sequentially on the host processor. Both the Interlagos and Sandy Bridge processors run neck and neck with each other, with the former being marginally faster. On the GPU side, the Radeon 7970 ranges from being neck and neck to 1.8x faster than the GTX 480. This is unsurprising given that the GTX 480 is now two generations old. In our largest dataset, the Radeon 7970 yielded over a 5x improvement over either CPU.

In figure 6.8, we compare Specmaster's performance to Myrimatch. We find that GPU acceleration yields nearly an order of magnitude in performance improvement using the Radeon 7970 in our largest dataset, compared to Myrimatch running on Sandy Bridge. Superficially, figures 6.9 and 6.8 may seem contradictory; the Sandy bridge processors are faster in the latter but slower in the former, and the GTX 480 appears faster than the Radeon 7970 in the latter while slower in the former. This result is testament to the bane of strong scaling.

Specmaster experiences the ramifications of Amdahl's law[44]; the peptide search itself now accounts for under half of the total application in the standard searches we performed. Nearly all of the remaining application time resides in peptide generation, which Specmaster performs sequentially on the CPU. For standard peptide searches without modifications (which are currently unsupported), further non-trivial attempts to accelerate Specmaster should focus on peptide generation, which we perform sequentially on the CPU (an exercise left to the reader). However, replacing a sequential loop with a parallel loop is a fairly trivial exercise. Furthermore, if Specmaster supports PTM search capabilities in a way that doesn't generate additional peptides, this will dramatically increase computational intensity per peptide. This will analogously reduce the fractional runtime that's sequential in Specmaster (i.e. exploit Gustafson's law[66]).

6.5 Parallel Results

Specmaster can additionally process scans in parallel using clUtil's ParallelFor function. Figure 6.10 shows the performance obtained under the static scheduler, the EGS scheduler,

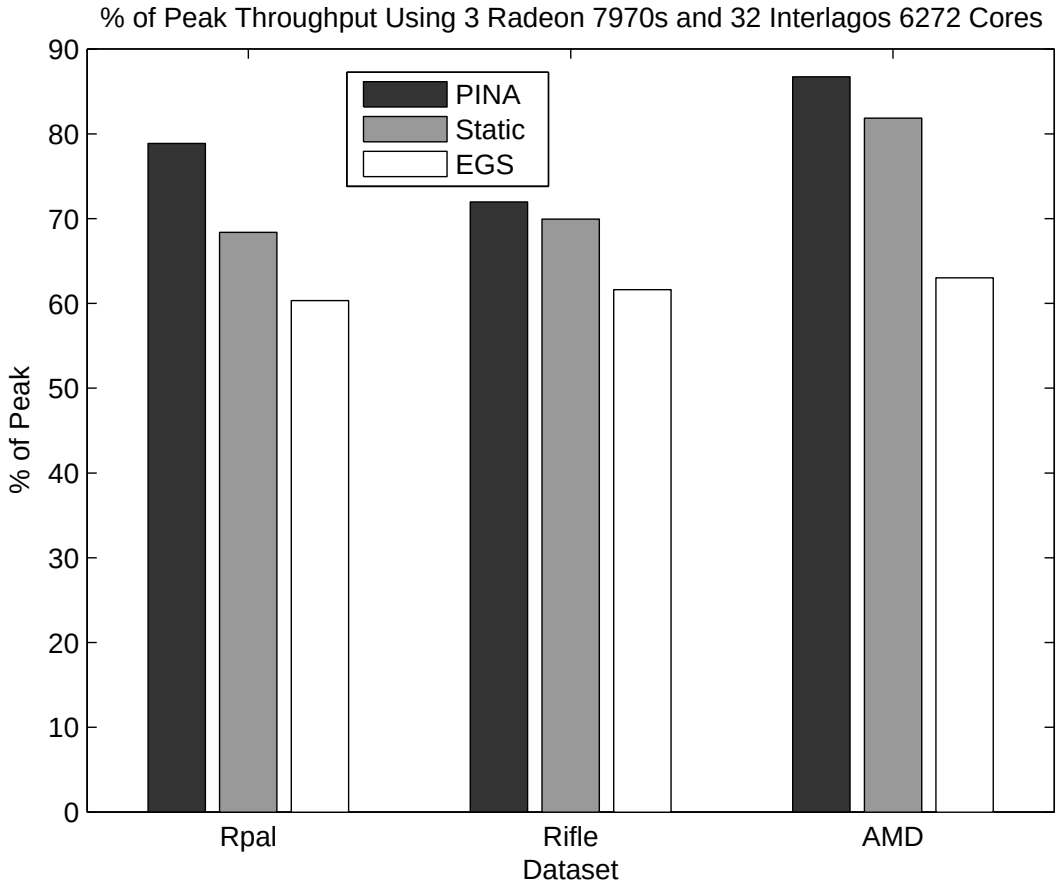

Figure 6.10: Specmaster peptide processing fraction of peak throughput using 3 Radeon 7970s and 32 Interlagos 6272 cores

and the PINA scheduler using 3 Radeon 7970s and the 32 Interlagos cores. The static scheduler in this application divides the iteration space into 30 chunks roughly equal in size. The PINA scheduler offers significant performance gains over EGS and modest gains over the static scheduler. Oddly enough, the EGS scheduler suffers from severe performance issues despite being designed with hybrid execution in mind.

Figure 6.11 lists the fraction of peak throughput obtained using 3 Radeon 7970s. Performance rankings and gains are similar to those in the hybrid experiment. Combining the results from these two experiments yields some interesting observations. Firstly, the performance issues with the EGS scheduler are systematic; it provides significantly less

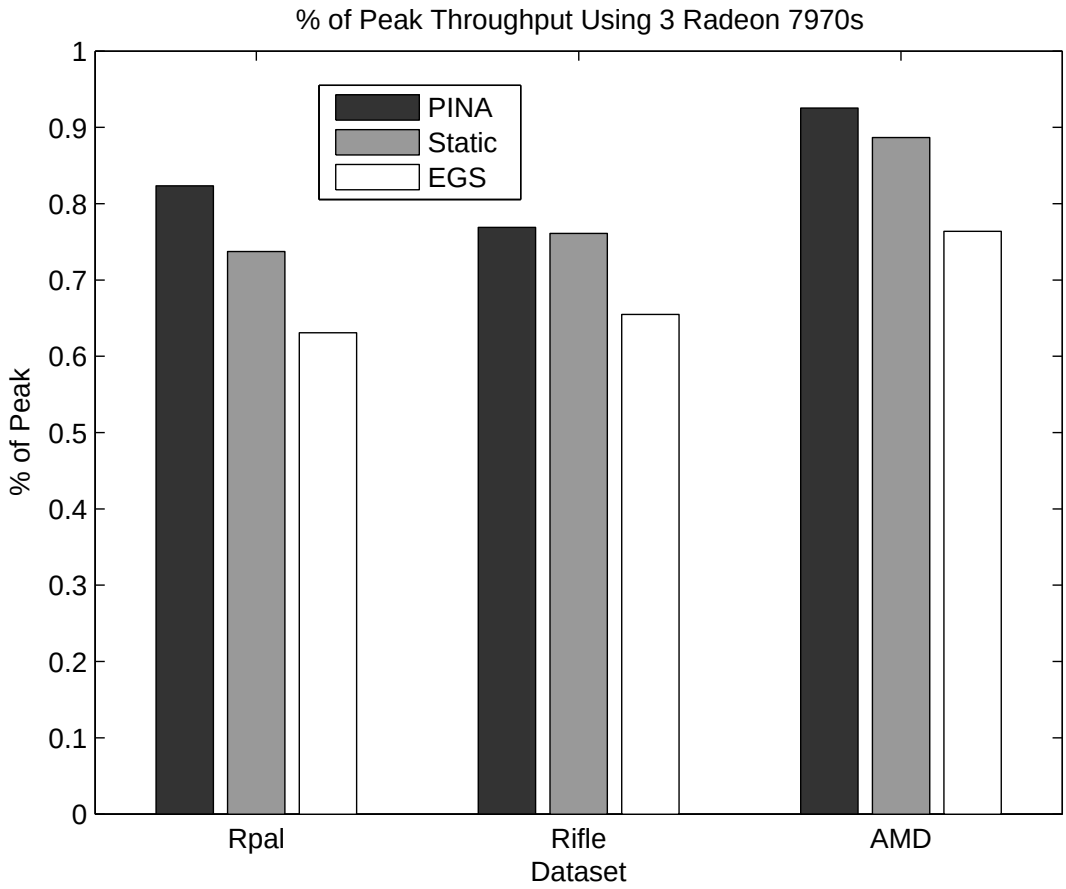

Figure 6.11: Specmaster peptide processing fraction of peak throughput using 3 Radeon 7970s

performance in every experiment. Secondly, the performance gap between the PINA and static schedulers widens under the heterogeneous experiments. This occurs because the PINA scheduler uses its autotuning model to assign appropriate amounts of work to both the CPU and GPU. The static scheduler on the other hand always assigns equal sized chunks to each device, either giving too much work to the CPU or too little to the GPU in the process. Finally, despite the PINA scheduler operating relatively better in a heterogeneous environment, it still handily outperforms the static and EGS schedulers in the homogeneous environment. This provides evidence that autotuning is an effective method for discovering appropriate chunk sizes for loops.

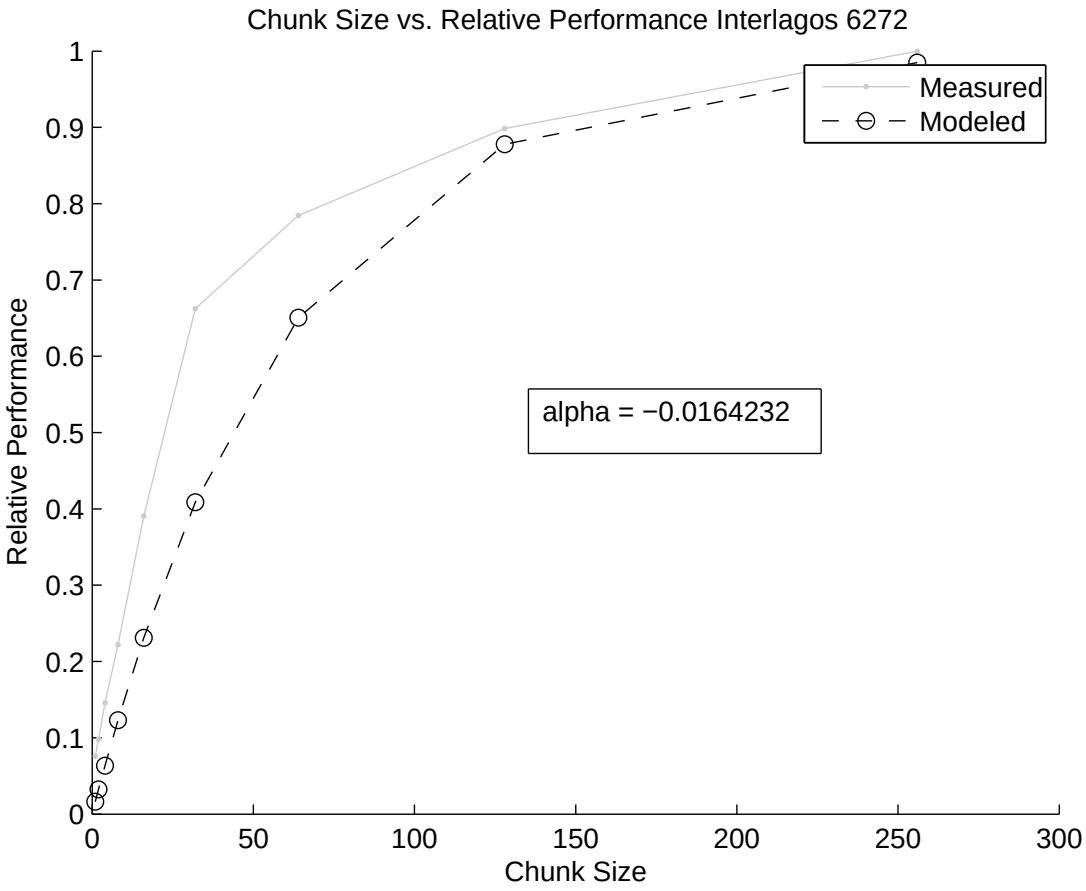

Figure 6.12: Autotuned performance model calculated from empirical data for the Interlagos 6272

Using the autotuning model works well in both the heterogeneous and homogeneous environments. Figures 6.12 and 6.13 show both the measured relative execution rates and the execution rates according to the PINA scheduler's derived model for the Interlagos 6272 and Radeon 7970. Specmaster uses a linear regression to compute α from the acquired data. Solving equation 6.4 with some fixed β (0.95 in these results) yields the chunk size for each device. In this case, the CPU has a desired chunk size of 182 and the GPU has a desired chunk size of 737.

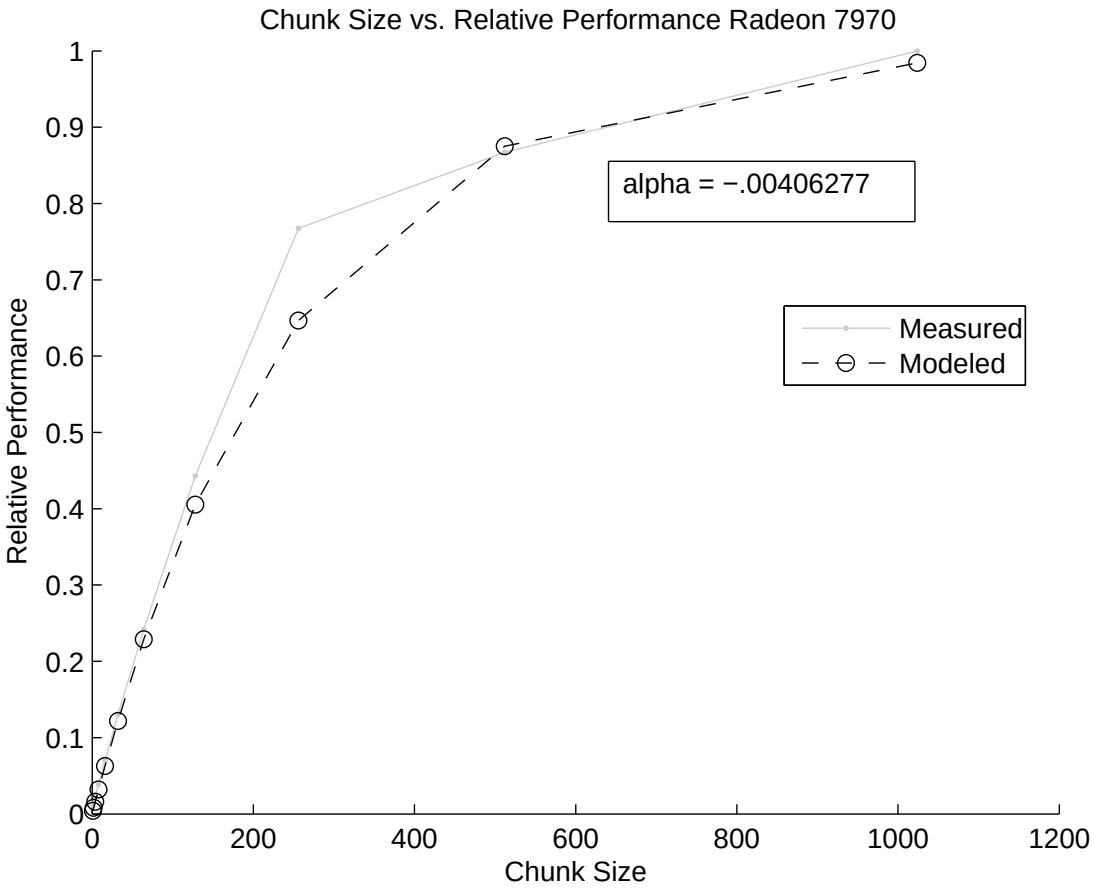

Figure 6.13: Autotuned performance model calculated from empirical data for Radeon 7970

$$chunk_{amortized} = \frac{\log(1-\beta)}{\alpha_d} \tag{6.4}$$

Using our clUtil library, we are able to provide multi-device execution with code that is significantly terser than vanilla OpenCL. clUtil manages hides of OpenCL's boilerplate code and reduces objects we instantiate to only include buffers residing on devices. Furthermore, using our ParallelFor loop, we achieved multi-device execution by replacing a single for loop. Specmaster serves as interesting datapoint due to its non-triviality.

While Specmaster's initial design took several months using vanilla OpenCL, we quickly obtained further performance gains and giant leaps in code quality using clUtil. We spent most of our development time on debugging the device kernels, which require a combination of black-box debugging and tedious emulation on the CPU. In its original form, Specmaster had very verbose host code for launching kernels, copying data between device and host, and buffer allocation. clUtil vastly streamlined this segment of the application and reduced code size and complexity while improving readability. This came at virtually unmeasurably small performance costs while allowing us to managably increase Specmaster's complexity through new features.

Specmaster's abilities to automatically tune kernels to different devices and distribute work to multiple devices would require an immense amount of additional vanilla OpenCL code and would likely make Specmaster unmaintainable for a single developer. However, using clUtil, we were able to not only obtain scalable performance in an aggressive heterogeneous system, we were able to optimize performance to better use each resource.

Chapter 7

Raytracing

7.1 Background

Raytracing is a technique that renders 3D graphics from first principles. As a result, many features that require complex approximations in traditional 3D renderers come freely with raytracing including: reflections, refraction, diffusion, specular lighting, shadows, and global illumination. Furthermore, Raytracing implementations can scale logarithmically as a function of the scene complexity when using bounding volume hierarchies or KD-trees[84]. This makes it suitable for applications that require large scenes and/or high accuracy. Figure 7.1 shows an example of a raytraced set of spheres above a plane.

7.1.1 Procedure

Turning any 3 dimensional scene into a 2D picture using any method amounts to determining the color of each pixel. True to its name, raytracing does this by casting *rays* from a point in space through a screen representing the picture, using one (or more with anti-aliasing) rays per pixel. In the simplest implementation possible, the algorithm identifies the first

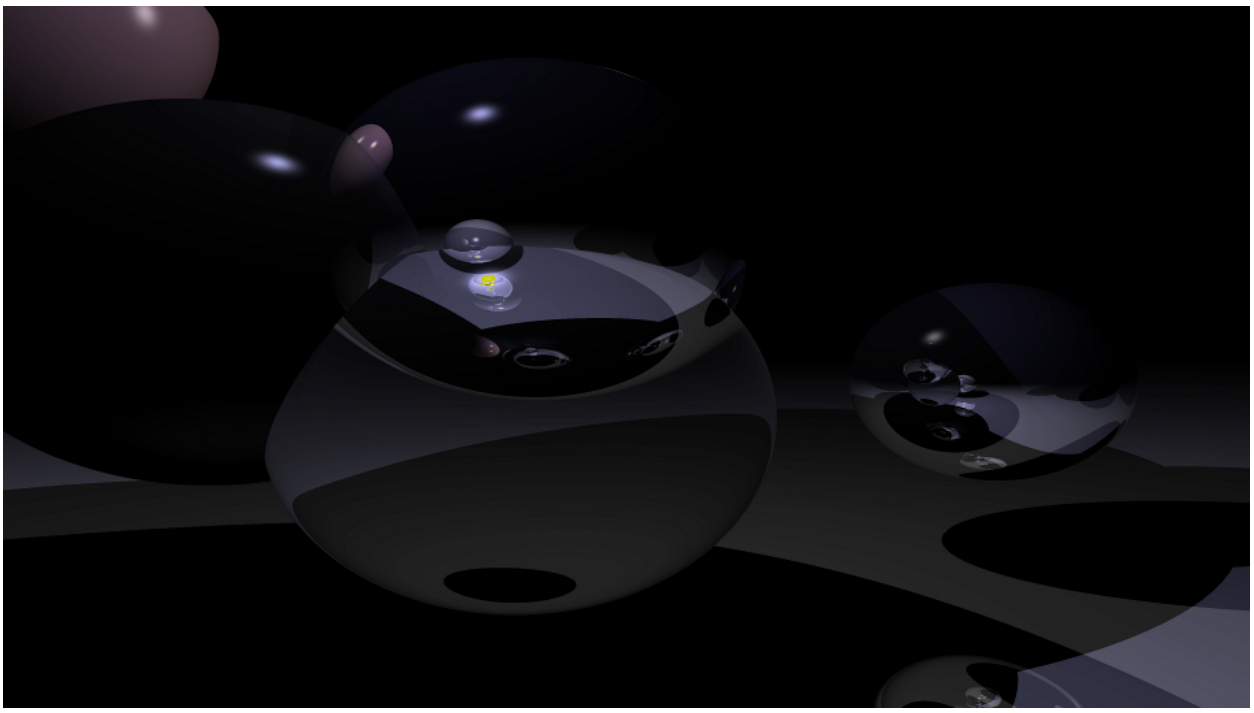

Figure 7.1: Raytraced group of spheres

object the ray intersects and assigns the objects color to the picture value. This yields flat projections of 3D objects that have no shading, depth, or lighting.

A still primitive, yet significant improvement over flat shading is diffusion shading. This technique uses Lambert's cosine law to compute the perceived color resulting from light bouncing off a surface (figure 7.2). Each object has a material property denoting the diffusion coefficient. Equation 7.1 shows how to compute the diffuse lighting contribution, c_d, where c_m is the color of the object R intersects, α_m is the diffusion constant for the intersecting object, N is the normalized vector normal to the object at the intersection point, L_i is a normalized vector in the direction of the i^{th} light, and c_i is the i^{th} light's color[85].

$$c_d = c_m \alpha_m \sum_i \vec{N} \cdot \vec{L_i} c_i \tag{7.1}$$

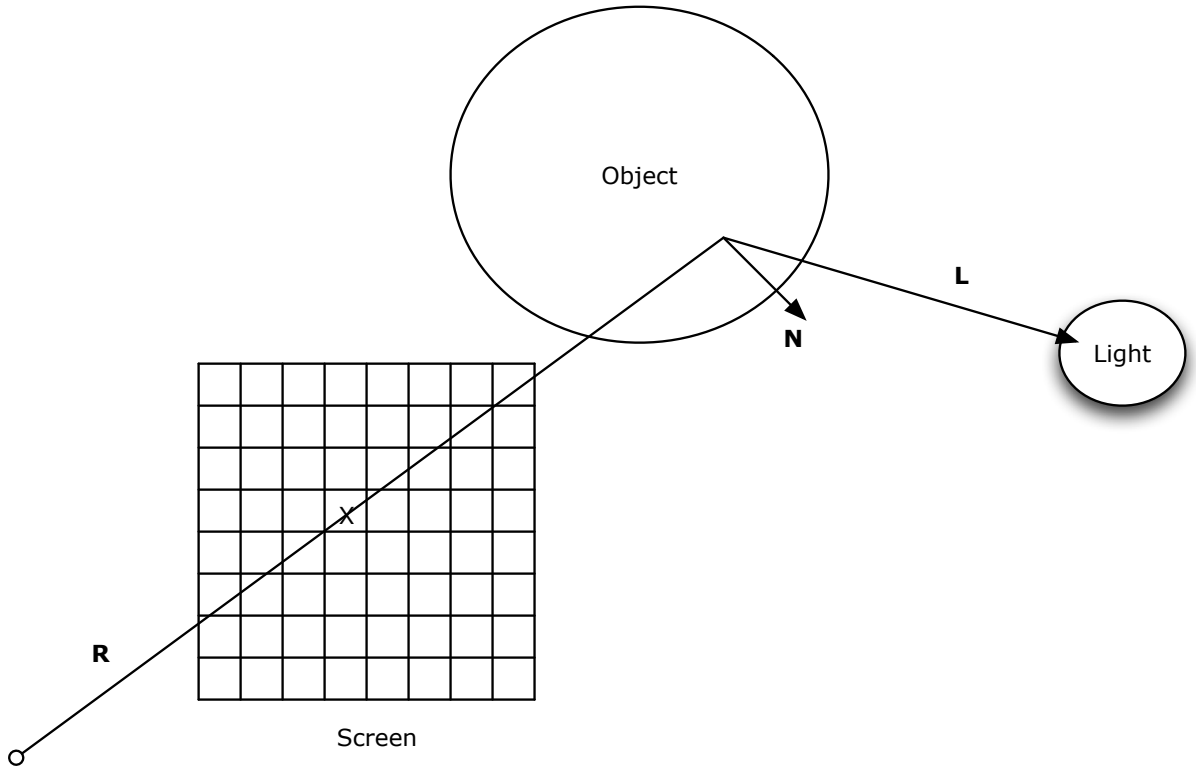

Figure 7.2: A ray (R) emitted through a pixel hitting the object. The diffuse lighting is a material property coefficient multiplied by the cosine of the surface normal (N) and the direction to the light (L)

Reflections add further realism and are a main selling point for raytracing. To implement reflections, simply spawn another ray at the point of intersection in the reflected direction. The reflected direction is given by equation 7.2 and behaves exactly like a primary ray, computing diffuse, specular, shadows, and even more reflections for the object it intersects[85]. In principle, rays can reflect ad infinitum, requiring infinite recursion. Arbitrarily stopping this process after a few reflections still yields a close approximation without an infinite runtime and stack size.

$$R' = 2(\vec{R} \cdot \vec{N})\vec{N} - \vec{R} \tag{7.2}$$

Refractions bend light as it passes through a translucent object. A simple example of this phenomena is the distorted view seen through a glass of water. Refractions are somewhat more complicated as the refraction ray's angle is a function of the refraction indices of the materials inside and outside of the intersecting object. These indices affect the way light bends. Equation 7.3 shows how to compute a refraction vector R'' [85]. In this equation, η_L is the index of refraction of the material light is leaving and η_T is the index of refraction light is entering, both of which are material constants.

$$R'' = \left(\frac{\eta_L}{\eta_T} \vec{N} \cdot \vec{L} - \sqrt{1 - \frac{\eta_L^2}{\eta_T^2} \left[1 - \left(\vec{N} \cdot \vec{L} \right)^2 \right]} \right) \vec{N} - \frac{\eta_L}{\eta_T} \vec{L} \tag{7.3}$$

Shadows and specular lighting provide the two final elements of the raytracer we used. When iterating over light sources to determine a ray's color, first check to see if other objects are in the way. If so, multiply by a value smaller than 1 to darken the color (the raytracer I'm using merely multiplies by zero). This removes an obscured light source's contribution to the ray's color. Finally, specular lighting provides shiny highlights for direct lighting.

7.1.2 Computational complexity and parallelism

Efficient raytracers can achieve $O(R\log(N))$ complexity, where R is the number of rays and N is the number of objects in a scene. Without complex bounding volume hierarchies or KD-trees, the runtime complexity is $O(RN)$. While unacceptable for real raytracing applications, this simple approach works sufficiently for demonstrating Raytracing's performance scalability. Since each pixel is independent, their corresponding rays can compute in parallel. Indeed, this is how we extract both coarse and fine grain parallelism. Our raytracer implements the $O(RN)$ algorithm, as the $O(R\log(N))$ algorithms feature tradeoffs and are active areas of research in themselves.

7.2 Relevant Work

The inclusion of Raytracing as a parallel for accelerated algorithm contributes little to the graphics field. Much of the related work is significantly more sophisticated, some of which already feature hybridized GPU-CPU implementations. Rather, we include raytracing to demonstrate the ease of integrating the parallel for loop into a pre-existing application as well as providing another data point for comparing the schedulers' performances.

There exist a number of notable ray tracing engines that exploit GPU acceleration. One of the earlier attempts uses DirectX and OpenGL [86] and finds little speedup over using the CPU as a result of the GPU's then primitive functionality. In [87], the authors implement a real-time raytracer that can render a scene with 12.7 million triangles at 1024x1024 resolution at 3FPS using CUDA. [88] provides a brief survey of Ray tracing on the GPU and provides a comparison of several of the acceleration hierarchies. There also exist OpenCL ray tracers such as SmallLuxGPU[89] which can load balance rendering on the GPU and CPU.

7.3 Implementation

In my work, we take an existing open source simple OpenCL renderer and modify it. Rather than rendering to an OpenGL context, this work renders to a bitmap file. Furthermore it leverages clUtil's ParallelFor() function to divide rendering onto multiple devices. This includes multi-GPU and hybrid CPU/GPU permutations. The existing raytracer* is a BSD licensed project for rendering simple scenes on OpenCL devices. We made a few modifications to both its front-end code and kernel to support multi-device execution.

The raytracing algorithm supports reflections, refractions, specular and diffuse lighting, and shadow rays. A hard-coded function assembles the scene (composed of spheres, planes, and lights) in each thread. The algorithm then traces a single ray per work item. Reflection and refraction are typically implemented using recursion, which is forbidden in OpenCL. To surmount this limitation, the kernel uses small arrays as manually managed stacks. This allows the emulation of recursion without needing to save function return addresses on an implicit stack.

The algorithm extracts both coarse and fine grain parallelism over separate rays emitted through the screen. The algorithm doesn't perform anti-aliasing, which means that for a w x h image, there exist wh independent rays. At the fine grain level, the algorithm parallelizes over both pixels within a scan line and separate scan lines. The course grain parallelism processes different batches of scanlines concurrently. clUtil's parallel for loop implements the coarse-grain parallelism (figure 7.3) while separate work items and work groups in a kernel implement the fine-grain parallelism.

Two changes to the kernel enabled it to process batches of rays as opposed to the entire scene (figure 7.4). Firstly, an additional rowOffset parameter allows the kernel to start at an arbitrary scanline instead of the first. Secondly, an if statement prevents the kernel from rendering outside of its designated work area. This is necessary because the local work group

*https://code.google.com/p/basic-opencl-raytracer/

```
ParallelFor(0, 1, height - 1, [&](size_t start, size_t end)
{
  size_t curDevice = Device::GetCurrentDeviceNum();
  unsigned int count = end - start + 1;
  unsigned int offset = (unsigned int) start;

  size_t localSize;
  size_t curQueue = 0;

  if(Device::GetDevices()[curDevice].getDeviceInfo().Type ==
      CL_DEVICE_TYPE_CPU)
  {
    localSize = 4;
  }
  else
  {
    localSize = 16;
  }

  for(size_t curRow = 0; curRow < count; curRow += 256)
  {
    Device::GetCurrentDevice().setCommandQueue(curQueue);

    unsigned int innerRowCount = count - curRow > 256 ?
      256 : count - curRow;
    unsigned int innerOffset = offset + curRow;

    clUtilEnqueueKernel("raytracer",
                        clUtilGrid(width, localSize, count, localSize),
                        *deviceBuffers[2 * curDevice + curQueue],
                        width,
                        height,
                        innerRowCount,
                        innerOffset);
#if 1
    deviceBuffers[2 * curDevice + curQueue]->
      get(&image[4 * innerOffset * width],
          4 * sizeof(float) * innerRowCount * width);
#endif

    curQueue = curQueue == 0 ? 1 : 0;
  }
});
```

Figure 7.3: Using clUtil's parallel for loop to extract coarse grained parallelism

```
__kernel void raytracer(__global uint *output, uint width, uint height)
{
  int2 pos = (int2)(get_global_id(0), get_global_id(1));
  float2 screen = (float2)(
                           pos.x / (float)width * 8.f - 4,
                           pos.y / (float)height * 6.f - 3
                          );

  struct Ray r;

  r.origin = (float4)(0, 0, -5, 0);
  r.dir    = normalize((float4)(screen.x, screen.y, 0, 0) - r.origin);

  //Build the scene...

  output[pos.x + pos.y * width] = rgbtouint(recursivetrace(&s, &r));
}
```

```
__kernel void raytracer(__global float4 *output, uint width,  uint height, uint count, uint rowOffset)
{
  int2 pos = (int2)(get_global_id(0), get_global_id(1));
  float2 screen = (float2)(
                           pos.x / (float)width * 8.f - 4,
                           (rowOffset + pos.y) / (float)height * 6.f - 3
                          );
  if(pos.x < width && pos.y < count)
  {
    struct Ray r;

    r.origin = (float4)(0, 0, -5, 0);
    r.dir    = normalize((float4)(screen.x, screen.y, 0, 0) - r.origin);

    //Build the scene...

    output[pos.x + pos.y * width] = recursivetrace(&s, &r);
  }
}
```

Figure 7.4: Top: unmodified raytracing kernel present in original source code Bottom: kernel modified to support working on pieces of the scene. Changes include the addition of a rowOffset parameter and a branch to ensure that the ray should actually be computed.

size can be an arbitrary amount, which in turn can create more work items than exist rays in a loop chunk.

The simplicity of the two modifications to the raytracing engine highlight the integration ease of clUtil's ParallelFor loop. The loop itself is under 50 lines of code and the kernel required minor changes to 5 lines of code. These small changes move the functionality from executing on a single device to using all devices available in the system.

7.4 Performance results

In principle, raytracing exhibits very good strong-scaling. There exist rays that can be divided among devices. Figure 7.5 shows the performance scalability using 1 to 32 cores. Interestingly enough, the performance drops most quickly moving from 1 to 2 cores. One

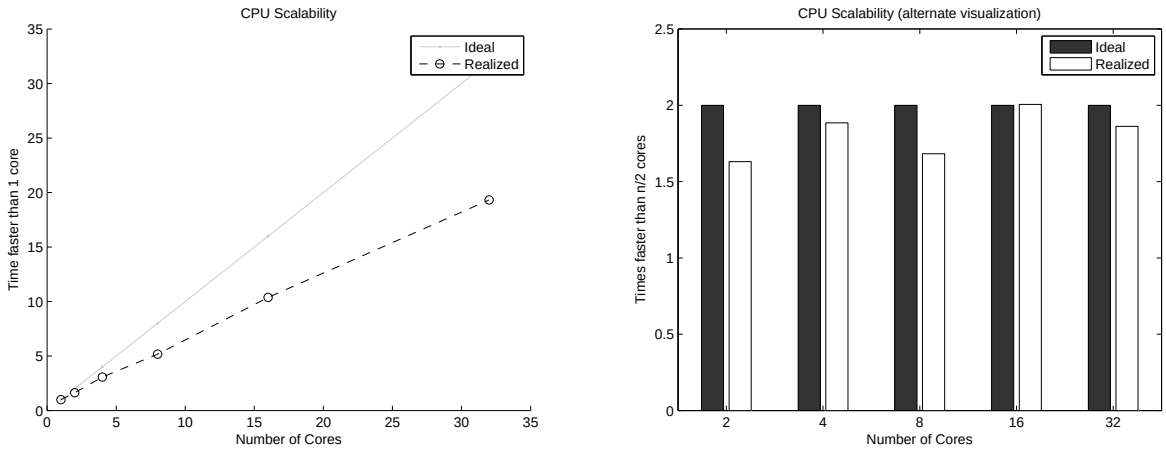

Figure 7.5: Left: raw speedup using 1, 2, 4, 8 ,16, and 32 cores. Right: alternate representation

possible reason for this is that Bulldozer processors can dynamically increase the clock frequency for higher performance in sequential programs. As the clock rate decreases as more cores become active, the ideal speedup is no longer a linear function of the number of cores being used. Since the performance degradation drops off as the number of cores continues to increase, raytracing does actually scale well to 32 cores.

All performance experiments render a simple scene of 7 spheres, a plane, and 2 lights at 2160p (3840 x 2160) HD resolution (figure 7.1). Since PCIe data transfers are uninteresting in understanding performance scalability, the data transfer back to host is omitted. The loop is designed to double pump execution such that transfers and computation can overlap, but even still the transfer represents a large burden for such a simple scene. More representative scenes with hundreds of thousands to millions of objects would provide ample computation to amortize the data transfer of the rendered pixels. Furthermore, when using only the GPUs, one could use Crossfire or SLI to share the rendering workload between GPUs and have the screen bitmap appear on a single device.

When using all 3 GPUs in the system, performance scales very close to the theoretical throughput using all three schedulers (figure 7.6). All three schedulers are above 85% of

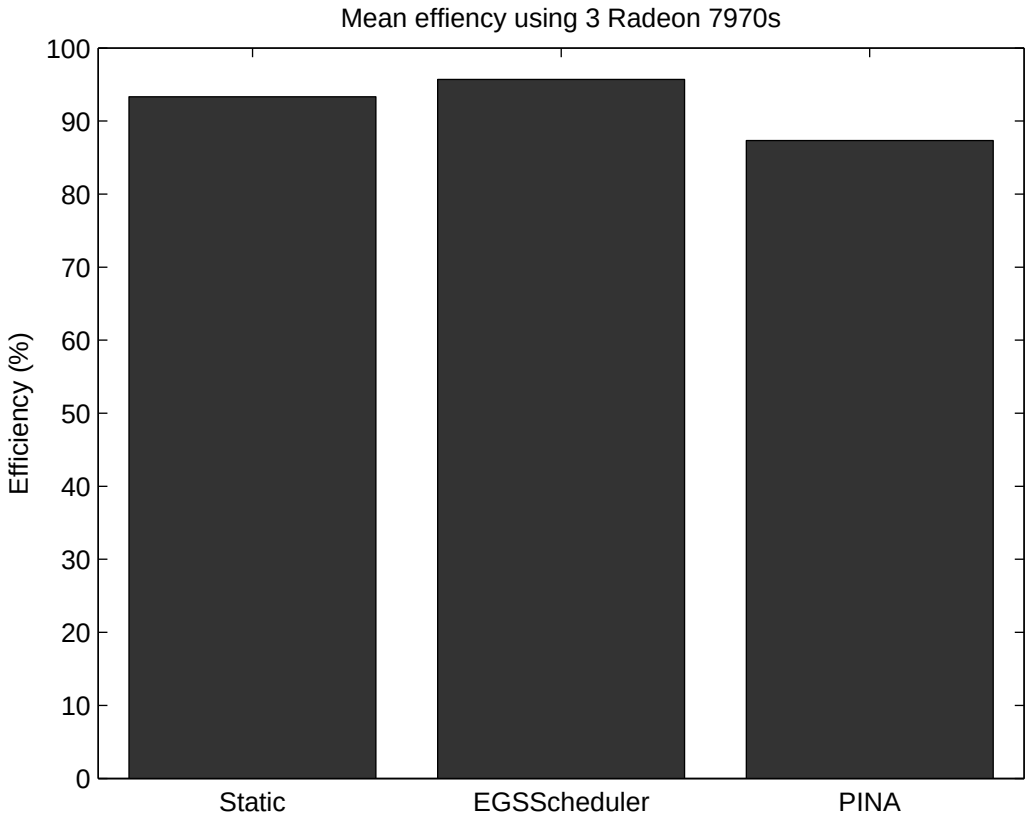

Figure 7.6: Scheduler efficiency using 3 Radeon 7970s

three times the throughput of a single Radeon 7970. Interestingly enough, the EGSScheduler performs most admirably, achieving 95% of the theoretical throughput. Outperforming the static scheduler in the homogeneous case would imply there exists workload imbalance in the iteration space. Indeed, black areas of the scene require very little execution time as there are no reflections, refractions, or even primary collisions to compute. The shiny spheres however create many derivative rays that require additional computation.

When using all 3 GPUs and all 32 cores, overall performance drops significantly over using only the GPUs. All schedulers suffer from poor efficiency (figure 7.7). EGS again performs the "best," though still significantly slower than using only the GPUs. In this situation, the PINA scheduler fails to even achieve 50% efficiency. Furthermore, the performance discrepancy between the schedulers is very large. This situation may seem puzzling given

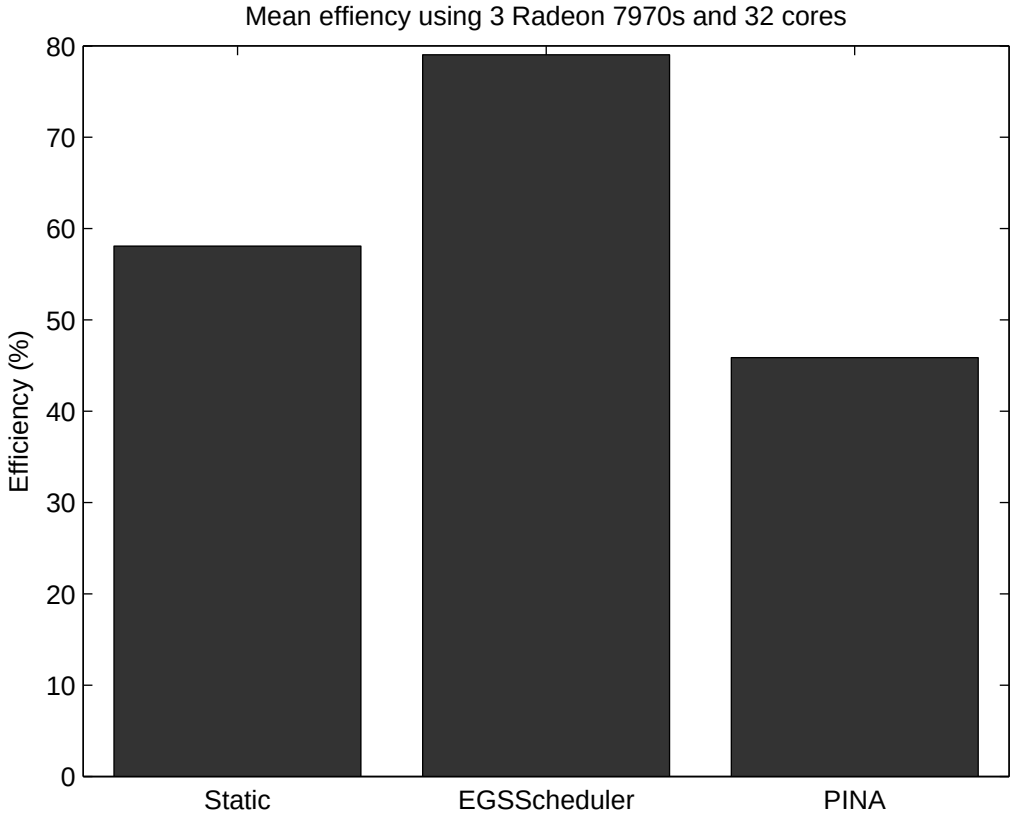

Figure 7.7: Scheduler efficiency using 3 Radeon 7970s and 32 cores

that problem scales well on both the CPU and GPU. However, resource contention is likely a huge problem when using all of the resources.

Experimentation reveals how many threads AMD's APP SDK spawns per device. Each device has management 2 threads (e.g. to execute data transfers) and the CPU device uses 2 compute threads per core. Running with 3 GPUs and all 32 cores requires 73 threads (counting the main thread). This can cause preemption problems; if the main thread gets preempted when a device is free, that device must wait until the OS reschedules the main thread to get work. Similarly, if a helper thread gets preempted while trying to launch a kernel, the device must again wait until the OS schedules the helper thread. Finally, even if all helper threads and the main thread are running, the CPU worker threads must necessarily not all be running (due to limited resources). This causes CPU kernels to execute

more slowly as they must wait for the OS to reschedule preempted workers to make forward progress. All three of these reasons translate to reduced performance.

Indeed, when running with all 3 GPUs and 16 cores, the heterogeneous performance is markedly improved (figure 7.8). When resource contention reduces, hybrid execution performance improves dramatically. The static scheduler poorly handles load imbalances caused by assigning equal work to each device type. The EGSScheduler performs well, but by design can give to little work to devices at the end as seen in Specmaster. The PINA scheduler achieves over 90% of the theoretical throughput attainable using 16 cores and 3 GPUs. Increasing the number of cores used to 17 showed an enormous performance drop. Should AMD reduce the number of threads needed to efficiently use a device, using more cores in a heterogeneous application may be more feasible.

Since our efficiency metric can hide actual performance after processing, we provide the real throughputs of the three schedulers in figure 7.9. Primary rays count only those emitted from the screen, of which there is one for every pixel in the 2160p image. This figure is easier to count than the total number of rays cast per second, which is data dependent. These numbers are for this simple scene only and should not be used to compare the performance of our suboptimal raytracer to others.

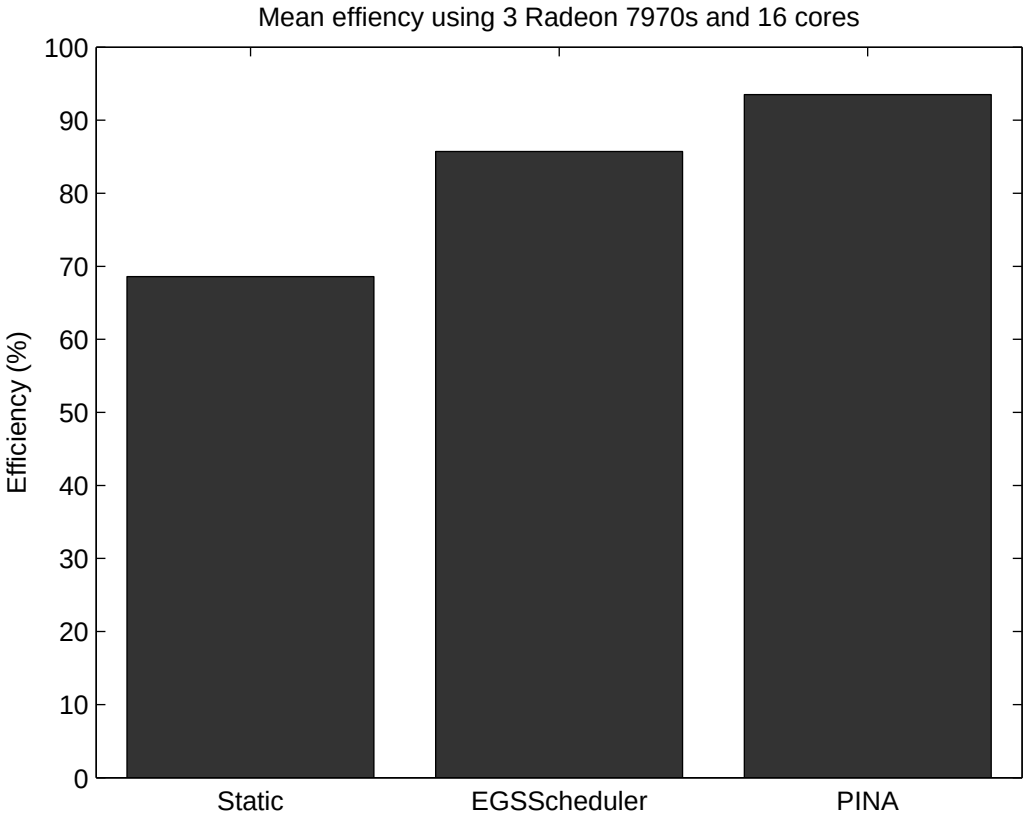

Figure 7.8: Scheduler efficiency using 3 Radeon 7970s and 16 cores

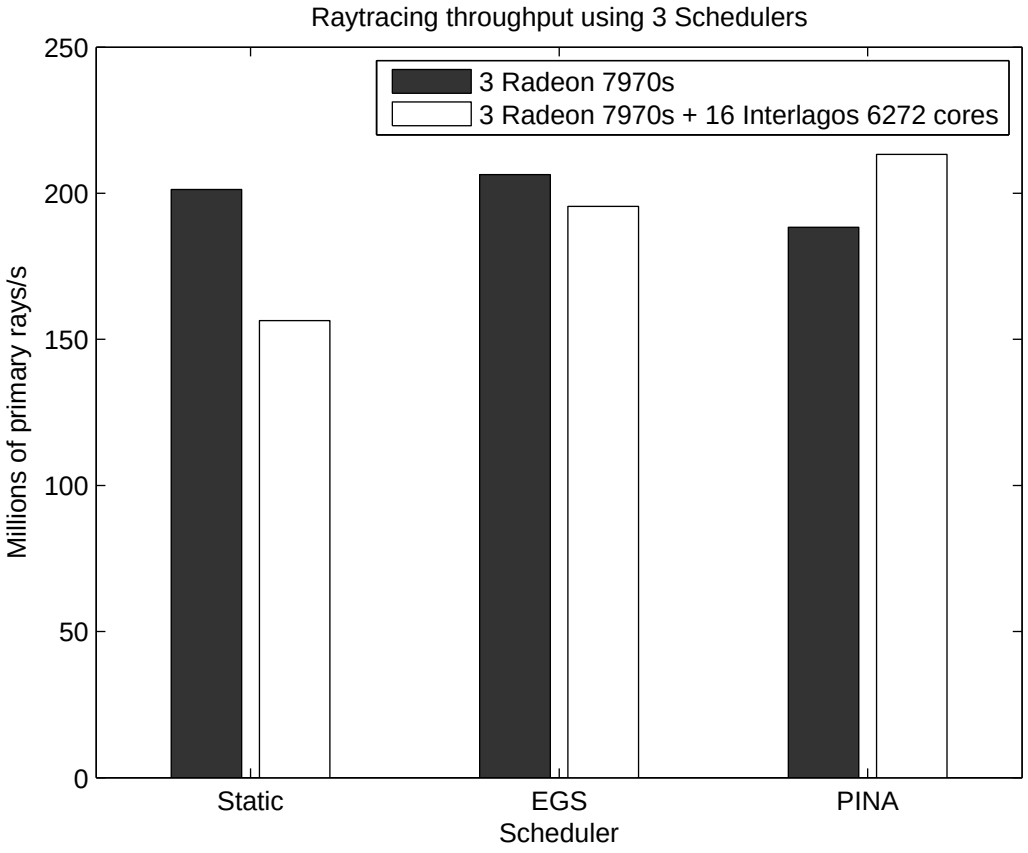

Figure 7.9: Scheduler throughputs in millions of primary rays per second

Chapter 8

Conclusions

This dissertation explored parallel loop structures running with multiple heterogeneous OpenCL devices using three different schedulers. clUtil's ParallelFor loop provides a familiar and simple interface for easily using multiple accelerators in tandem. Its flexibility allows one to select one of three loop schedulers including the novel PINA loop scheduler. This scheduler leverages autotuning and a predictive model to provide better acceleration over the other two tested arbiters in two out of three tested applications and nearly identical performance in GEMM.

The key feature of the PINA scheduler, autotuning, uses a novel approximation to Gustafson's law. This approximation vastly simplifies empirically modeling sequential work amortization and readily predicts the relative speedup a device should yield given a parallel workload. This approximation represents the most important theoretical contribution, while clUtil, the schedulers, and the parallel for loop implementation represent technology contributions in the largely untapped (and increasingly important) field of heterogeneous computing using accelerators.

There exist a number of logical improvements to the PINA scheduler that remain as future work. Firstly, the scheduler can use its predictive model to automatically disable devices that yield little performance benefit. This approach is similar to the work in [59]. Secondly,

the PINA scheduler can trade execution efficiency for better load balancing. For example, if one chunk executes at 95% efficiency or 66% efficiency as two chunks, the runtime can split the chunk to attain better load balancing. This may entail using its iteration runtime model to predict the likelihood of multiple devices becoming free in some time quanta. Finally, the amortization model can improve its accuracy by removing more of its tail as discussed in chapter 4.

A number of architectural and application issues can limit PINA's performance. Ideally, slower devices require less work to amortize kernel launches. This allows for better load balancing as overall progress doesn't wait on slower devices (a scenario seen in the GEMM application). Disabling devices or running with reduced efficiency are two approaches that can help improve load balancing and overall efficiency at the expense of a single device's efficiency. Exploring the exact details of this remains as future work.

Heavy resource contention is another issue that needs to be addressed in the future. As shown in the raytracing application, while the application scaled well on the CPU alone, using both the CPUs and GPUs resulted in poor performance. This results from the dual role of the CPU serving as both a kernel execution device and a resource scheduler. Addressing this issue will likely require more rigorous examination of resource contention and may even be application dependent.

A logical extension of this work is heterogeneous distributed systems. Imagine a supercomputer where each node contains different numbers and types of devices. Because the parallel for loop methods presented in this paper don't assume shared memory (i.e. a GPU can't necessarily or efficiently access the CPU's main memory), extending this work to an distributed system is fairly straightforward in principle. One may note that iteration chunks are recursively divisible; a hierarchy of schedulers can treat progressively larger pools of machines as devices. The scheduler then breaks its iterations into smaller pieces applying one of the scheduling algorithms to dole work out to its subset of machines. Eventually, an individual node receives a chunk of work, at which point the multi-device parallel for methods discussed in this dissertation perform the actual work. To work in the general case,

each node should have a copy of all data, as it doesn't know a priori which iterations it will compute (barring complex communication and data locality tracking).

This method also should be amenable to fault tolerance with some work on the user's behalf. A first pass approach to this would be to have each node should compute a hash of its outputs and computing each iteration block twice. If the hashes are the same, then an error most likely did not occur. If the hashes do not agree, rerun the iterations a third time and vote.

Other interesting extensions include minimizing energy instead of time. In principle, one could measure energy instead of time and compute iterations on the device that yields the smallest energy expenditures. Furthermore, PINA could measure energy amortization rather than time.

Because of parallel for's versatility and simplicity, it remains relevant even in the heterogeneous era for a variety of applications. However, it is by no means the only parallel paradigm. Another interesting area of research includes DAG scheduling with data locality awareness. clUtil already tracks data dependencies, making DAG scheduling a logical next step.

8.1 Contributions

The work presented in this dissertation provides a powerful library for writing software on heterogeneous systems. clUtil provides simple constructs that immensely improve programmer productivity by abstracting away boilerplate code, removing handles, improving syntax, and replacing verbose error checking code with exceptions. We trivially parallelized three applications to run on multiple GPUs (and even the CPU in hybrid execution) with existing kernels while achieving good scalability.

In matrix multiplication, we took an the existing GEMM kernel and used this as the basis for a larger matrix multiply. With clUtil's ParallelFor loop and it's simplified programming

model, we were able to create once of the fastest single-machine GEMM algorithms to date in roughly 150 lines of driver code. This algorithm overlaps computation and communication and efficiently uses an arbitrary number of GPUs.

In our second application, we took our existing vanilla OpenCL peptide search engine and improved maintainability while further optimizing performance using clUtil. Our library enabled us to dramatically reduce the number of object handles and memory leaks associated with improperly calling OpenCL functions to free objects. Furthermore, clUtil significantly reduces the number of calls and parameters required to achieve the same functionality as a vanilla OpenCL implementation. Using clUtil's ParallelFor loop, we attained a near-linear speedup with 3 GPUs by changing roughly 5 lines of code. We used the savings in complexity to create device-specific optimizations, making Specmaster a fairly sophisticated application that truly exploits the promise of OpenCL.

In our final application, we rewrote the existing OpenCL driver code using clUtil. Our version was significantly terser, easier to follow, and more functional, as it executes on an arbitrary number of OpenCL devices.

We hope that clUtil will increase interest in heterogeneous computing by lowering the barriers of entry. CUDA has dominated the GPU research arena since its inception. While OpenCL can in principle support many devices and offers portability, most researchers today continue to use CUDA in part due to OpenCL's complexity. Since clUtil removes this limitation and actually improves on CUDA, we hope to see more applications written end-to-end to use arbitrary numbers of devices and device types as Specmaster does. The work presented in this dissertation serves as a practical how-to guide for writing portable and performant code that can target a multitude of devices.

Bibliography

[1] M. D. Ercegovac, "Heterogeneity in supercomputer architectures," *Parallel Computing*, vol. 7, no. 3, pp. 367 – 372, 1988. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167819188900555 1

[2] H. Nicholas, G. Giras, V. Hartonas-Garmhausen, M. Kopko, C. Maher, and A. Ropelewski, "Distributing the Comparison of DNA and Protein Sequences Across Heterogeneous Supercomputers," in *Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on*, Nov 1991, pp. 139 –146. 1

[3] A. Khokhar, V. Prasanna, M. Shaaban, and C.-L. Wang, "Heterogeneous supercomputing: Problems and issues," in *Heterogeneous Processing, 1992. Proceedings. Workshop on*, Mar 1992, pp. 3 –12. 1

[4] J. Markoff, "The Attack of the 'Killer Micros'," *New York Times*, May 1991. 2

[5] T. H. Dunigan, "Beta Testing the Intel Paragon MP," ORNL, Tech. Rep., Jun 1995. 2

[6] Top 500 Supercomputer Sites, "T3D MC1024-8," http://www.top500.org/system/1404, 1994. 2

[7] Top 500 Supercomputing Sites, "CM-5/896," http://www.top500.org/system/1747, 1994. 2

[8] Algotrinix Ltd, "The CHS 2x4: The World's First Custom Computer," 1990. 2

[9] A. J. v. d. Steen, "FPGA-based Accelerators," http://www.phys.uu.nl/~steen/web08/fpga-accel.html, 2008. 2

[10] R. Hartenstein, "The Kress-Kung Machine Page," http://anti-machine.org/, 2001. 2

[11] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 203 –215, feb. 2007. 2

[12] D. W. Wall, "Limits of Instruction-level Parallelism," 1991, pp. 176–188. 2

[13] P. P. Gelsinger, "Power play," *Commun. ACM*, 2002. 2

[14] S. A. McKee, "Reflections on the Memory Wall," in *Proceedings of the 1st conference on Computing frontiers*, ser. CF '04. New York, NY, USA: ACM, 2004, pp. 162–. [Online]. Available: http://doi.acm.org/10.1145/977091.977115 2

[15] F. J. Pollack, "New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies (keynote address)(abstract only)," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 32. Washington, DC, USA: IEEE Computer Society, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=320080.320082 2

[16] B. Schauer, "Multicore Processors - A Necessity," http://www.csa.com/discoveryguides/multicore/review.pdf, 2008. 3

[17] M. Funk, "Simultaneous Multi-Threading on eServer iSeries POWER5," http://www.ibm.com/systems/resources/systems_i_advantages_perfmgmt_pdf_SMT.pdf, 2004. 3

[18] Sun Microsystems, "UltraSPARC T2 Processor System On a Chip," http://wikis.sun.com/download/attachments/31400118/N2_Announce_Breakout_final.pdf, 2007. 3

[19] Intel, "Intel Hyper-Threading Technology," http://www.intel.com/technology/platform-technology/hyper-threading/. 3

[20] IBM, "IBM zEnterprise 196 (z196) Overview," http://www-03.ibm.com/systems/z/hardware/zenterprise/z196.html. 3

[21] Intel, "Intel Atom Processors," http://www.intel.com/content/www/us/en/processors/atom/atom-processor-embedded-technology.html. 3

[22] IBM, "Cell Broadband Engine Architecture and its First Implementation: A Performance View," http://www.ibm.com/developerworks/power/library/pa-cellperf/. 3

[23] Playstation University, "IBM Cancels Cell Processor Development," http://www.psuni.com/ibm-cancels-cell-processor-development-1295/, 2009. 3

[24] Intel, "iSBX 275 Video Graphics Controller Multimodule Board Reference Manual," 1982. 4

[25] NVIDIA, "GeForce3," http://www.nvidia.com/page/geforce3.html. 4

[26] A. L. Shimpi, "ATI's Radeon 9700 (R300) - Crowning the New King," http://www.anandtech.com/show/947, 2002. 4

[27] Ádám Moravánszky and N. Ag, "Dense Matrix Algebra on the GPU," in *In Direct3D ShaderX2, Engel W. F., (Ed.). Wordware Publishing.* NovodeX AG, 2003, p. 2. 4, 41

[28] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM TRANSACTIONS ON GRAPHICS*, vol. 23, pp. 777–786, 2004. 4

[29] NVIDIA, "CUDA," http://www.nvidia.com/object/cuda_home_new.html. 4, 54

[30] V. Volkov and J. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, Nov 2008, pp. 1 –11. 4, 41

[31] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects," vol. 180. 4

[32] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, "Comparing Hardware Accelerators in Scientific Applications: A Case Study," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 58–68, 2011. 4

[33] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande, "Accelerating Molecular Dynamic

Simulation on Graphics Processing Units," *J. Comput. Chem.*, vol. 30, no. 6, pp. 864–872, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1002/jcc.21209 4

[34] N. Goodnight, "CUDA/OpenGL Fluid Simulation," 2007. 4

[35] AMD, "AMD 'Close to Metal' Technology Unleashes the Power of Stream Computing," http://www.amd.com/us/press-releases/Pages/Press_Release_114147.aspx, 2006. 4

[36] Khronos, "OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems," http://www.khronos.org/opencl/. 5

[37] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos, "Synthesis of Platform Architectures from OpenCL Programs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, May 2011, pp. 186 –193. 5

[38] D. Singh, "Higher Level Programming Abstractions for FPGAs Using OpenCL," http://www.eecg.toronto.edu/~jayar/fpga11/Singh_Altera_OpenCL_FPGA11.pdf. 5, 54

[39] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Computing*, no. 0, pp. –, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001335 5, 42, 43

[40] "OpenCL Bitcoin Miner," https://github.com/tcatm/oclminer. 5

[41] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, "A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators LAPACK Working Note 223." 5, 12

[42] NAMD, "Running NAMD:CUDA GPU Acceleration," http://www.ks.uiuc.edu/Research/namd/2.8/ug/node83.html. 5

[43] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving Dense Linear Systems on Platforms with Multiple Hardware Accelerators," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1504176.1504196 5, 41, 49

[44] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560 6, 75

[45] T. Casavant and J. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *Software Engineering, IEEE Transactions on*, vol. 14, no. 2, pp. 141 –154, Feb 1988. 9

[46] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, "Loop Transformations: Convexity, Pruning and Optimization ," in *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*. Austin, TX: ACM Press, Jan. 2011, pp. 549–562. 9

[47] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta, "GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation," in *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italy, Jan. 2010. [Online]. Available: http://hal.inria.fr/inria-00551516/en/ 9

[48] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007. 9

[49] Intel, "Intel Threading Building Blocks for Open Source," http://threadingbuildingblocks.org, 2011. 9

[50] A. N. Laboratories, "Message Passing Interface," http://www.mcs.anl.gov/research/projects/mpi/. 10

[51] M. Cierniak, W. Li, and M. J. Zaki, "Loop Scheduling for Heterogeneity," in *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, ser. HPDC '95. Washington, DC, USA: IEEE Computer Society, 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=822081.823033 10, 11, 16, 24, 25

[52] W. chung Shih, C. tung Yang, and S. shyong Tseng, "A Parallel Loop Self-Scheduling on Grid Computing Environments," in *Proceedings of the 2004 International Symposium on Parallel Architectures, Algorithms and IEEE Networks*, 2004, pp. 409–414. 12, 13, 25, 48

[53] T. Tzen and L. Ni, "Trapezoid Self-Scheduling: a Practical Scheduling Scheme for Parallel Compilers," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 4, no. 1, pp. 87 –98, Jan 1993. 12, 16

[54] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: a Method for Scheduling Parallel Loops," *Commun. ACM*, vol. 35, pp. 90–101, August 1992. [Online]. Available: http://doi.acm.org/10.1145/135226.135232 12

[55] C. D. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *IEEE Trans. Comput.*, vol. 36, pp. 1425–1439, December 1987. [Online]. Available: http://dx.doi.org/10.1109/TC.1987.5009495 12

[56] C. tung Yang and S. chyi Chang, "A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters," in *Proc. of Intl Conf. on Computational Science*. Springer-Verlag, 2003, pp. 1079–1088. 13, 25, 48

[57] C.-C. Wu, L.-T. Huang, L.-F. Lai, and M.-L. Chen, "Enhanced Parallel Loop Self-Scheduling for Heterogeneous Multi-core Cluster Systems," *Parallel Architectures,*

Algorithms, and Networks, International Symposium on, vol. 0, pp. 568–573, 2009. 13, 16, 20, 25, 39, 48, 49

[58] W.-C. Shih, C.-T. Yang, P.-I. Chen, and S.-S. Tseng, "A Hybrid Parallel Loop Scheduling Scheme on Heterogeneous PC Clusters," *Parallel and Distributed Computing Applications and Technologies, International Conference on*, vol. 0, pp. 56–58, 2005. 13, 16, 25, 48

[59] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro, "Predictive Runtime Code Scheduling for Heterogeneous Architectures," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 19–33. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92990-1_4 13, 14, 15, 16, 25, 48, 52, 95

[60] F. W. Burton and M. R. Sleep, "Executing functional programs on a virtual tree of processors," in *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, ser. FPCA '81. New York, NY, USA: ACM, 1981, pp. 187–194. [Online]. Available: http://doi.acm.org/10.1145/800223.806778 15

[61] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999. 15, 16

[62] W.-M. Hwu, Ed., *GPU Computing Gems Jade Edition*. Morgan Kaufmann, 2012. 15

[63] K. Spafford, J. Meredith, and J. Vetter, "Maestro: data orchestration and tuning for opencl devices," in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, ser. Euro-Par'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 275–286. [Online]. Available: http://dl.acm.org/citation.cfm?id=1885276.1885305 15, 16

[64] M. Roberts, *Signals and Systems: Analysis Using Transform Methods and MATLAB*, ser. McGraw-Hill Higher Education. McGraw-Hill, 2004. [Online]. Available: http://books.google.com/books?id=c2oN_GozNPoC 28

[65] R. L. Graham and R. L. Grahamt, "Bounds on Multiprocessing Timing Anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, pp. 416–429, 1969. 29

[66] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, pp. 532–533, 1988. 32, 75

[67] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, Sep. 1979. [Online]. Available: http://dx.doi.org/10.1145/355841.355847 40

[68] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers," http://www.netlib.org/benchmark/hpl/, 2008. 40

[69] Top500, "Top 500 supercomputing sites," http://top500.org, 2012. 40

[70] ——, "Tinahe-1a," http://i.top500.org/system/176929, 2012. 41

[71] ——, "Nebulae," http://i.top500.org/system/176819, 2012. 41

[72] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *CONFERENCE ON HIGH PERFORMANCE NETWORKING AND COMPUTING*. IEEE Computer Society, 1998, pp. 1–27. 41

[73] R. Nath, S. Tomov, and J. Dongarra, "An Improved Magma Gemm For Fermi Graphics Processing Units," *International Journal of High Performance Computing Applications*, vol. 24, no. 4, pp. 511–515, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1177/1094342010385729 42

[74] N. Nakasato, "A fast GEMM implementation on the cypress GPU," *Sigmetrics Performance Evaluation Review*, vol. 38, pp. 50–55, 2011. 42

[75] J. Fang, A. Varbanescu, and H. Sips, "A comprehensive performance comparison of cuda and opencl," in *Parallel Processing (ICPP), 2011 International Conference on*, sept. 2011, pp. 216 –225. 42

[76] N. Nakasato, "DGEMM Tahiti Update," http://galaxy.u-aizu.ac.jp/trac/note/blog/DGEMM_Tahiti_Update, 2012. 43

[77] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html 48

[78] R. Craig and R. C. Beavis, "TANDEM: matching proteins with tandem mass spectra." *Bioinformatics (Oxford, England)*, vol. 20, no. 9, pp. 1466–1467, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/bth092 53

[79] J. K. Eng, A. L. McCormack, and J. R. Yates, "An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database," *Journal of the American Society for Mass Spectrometry*, vol. 5, no. 11, pp. 976–989, Nov. 1994. [Online]. Available: http://dx.doi.org/10.1016/1044-0305(94)80016-2 53

[80] D. L. Tabb, C. G. Fernando, and M. C. Chambers, "MyriMatch:A Highly Accurate Tandem Mass Spectral Peptide Identification by Multivariate Hypergeometric Analysis," *Journal of Proteome Research*, vol. 6, no. 2, pp. 654–661, 2007, pMID: 17269722. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/pr0604054 53, 54, 71

[81] M. Brosch, L. Yu, T. Hubbard, and J. Choudhary, "Accurate and sensitive peptide identification with mascot percolator." *Journal of Proteome Research*, vol. 8, no. 6, pp. 3176–3181, 2009. [Online]. Available: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2734080&tool=pmcentrez&rendertype=abstract 53

[82] L. A. Baumgardner, A. K. Shanmugam, H. Lam, J. K. Eng, and D. B. Martin, "Fast Parallel Tandem Mass Spectral Library Searching Using GPU Hardware

Acceleration," *Journal of Proteome Research*, vol. 0, no. 0, 0000. [Online]. Available: http://dx.doi.org/10.1021/pr200074h 54

[83] B. Faherty, J. Milloy, and S. A. Gerber, "gMacro: GPU-CPU Computing for High Throughput Peptide Spectral Matching," Poster at American Society of Mass Spectrometry Annual Conference 2011. 54

[84] J. Arvo and D. Kirk, "A survey of ray tracing acceleration techniques," in *An introduction to ray tracing*. London, UK, UK: Academic Press Ltd., 1989, pp. 201–262. [Online]. Available: http://portal.acm.org/citation.cfm?id=94794 81

[85] J. Flynt, *Math for 3D Game Programming and Computer Graphics*, 3rd ed. Independence - Course Technology, CENGAGE Learning Distributor, 2011. 82, 84

[86] M. Christen, "Ray tracing on gpu," Ph.D. dissertation, University of Applied Sciences Basel, 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.98.1905&rep=rep1&type=pdf 85

[87] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on GPU with BVH-based packet traversal," in *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, Sep. 2007, pp. 113–118. 85

[88] M. Zlatuška, "Ray tracing on a gpu with cuda comparative study of three algorithms," *Electrical Engineering*, pp. 2–8, 2010. [Online]. Available: http://www.cgg.cvut.cz/members/havran/ARTICLES/ZlatuskaHavran2010wscg.pdf 85

[89] LuxRender, "Luxrender and opencl," http://www.luxrender.net/wiki/index.php?title=Luxrender_and_OpenCL. 85

Appendix

Appendix A

clUtil ParallelFor implementation

```
void clUtil::ParallelFor(const size_t start,
                         const size_t stride,
                         const size_t end,
                         function<void (size_t, size_t)> loopBody,
                         IScheduler&& model)
{
  size_t oldDeviceNum = Device::GetCurrentDeviceNum();
  size_t iterationsRemaining = end - start + 1;

  IndexRange range;
  range.Start = start;
  range.End = end;

  model.setRange(range);

  //Initialize device statuses
  vector<DeviceStatus> deviceStatuses(Device::GetDevices().size());

  for(size_t curDeviceID = 0;
      curDeviceID < deviceStatuses.size();
      curDeviceID++)
  {
    deviceStatuses[curDeviceID].DeviceID = curDeviceID;
  }

  //Parallel for scheduling loop
  while(iterationsRemaining > 0)
  {
    for(size_t curDeviceID = 0;
        curDeviceID < Device::GetDevices().size();
        curDeviceID++)
    {
      DeviceStatus& curDeviceStatus = deviceStatuses[curDeviceID];
      size_t deviceGroup = DeviceGroupInfo::Get()[curDeviceID];

      //If this device isn't busy, get some work from the model and run it
      if(curDeviceStatus.IsBusy == false &&
         model.workRemains(deviceGroup) == true)
      {
        IndexRange work;

        work = model.getWork(deviceGroup);

        Device::SetCurrentDevice(curDeviceID);
        Device& curDevice = Device::GetCurrentDevice();

        curDeviceStatus.Range = work;
        curDeviceStatus.Time1 = getTime();
        curDeviceStatus.IsBusy = true;

        loopBody(work.Start, work.End);

        //We indicate this device is finished by enqueueing markers into
```

```
        //every queues, then enqueueing a waitForEvents which depends on the
        //markers. Then enqueue one more marker so we can capture its event
        size_t prevCommandQueue = curDevice.getCommandQueueID();

        unique_ptr<cl_event[]>
          markerList(new cl_event[curDevice.getNumCommandQueues()]);

        for(size_t curQueueID = 0;
            curQueueID < curDevice.getNumCommandQueues();
            curQueueID++)
        {
          curDevice.setCommandQueue(curQueueID);

          clEnqueueMarker(curDevice.getCommandQueue(),
                          &markerList[curQueueID]);
        }

        curDevice.setCommandQueue(prevCommandQueue);

        clEnqueueWaitForEvents(curDevice.getCommandQueue(),
                               curDevice.getNumCommandQueues(),
                               markerList.get());

        clEnqueueMarker(curDevice.getCommandQueue(),
                        &curDeviceStatus.WaitEvent);

        //Release the horses from the gates
        curDevice.flush();
      }

      if(curDeviceStatus.WaitEvent != NULL) //If device has a valid event...
      {
        //Poll the event for completion
        cl_int eventStatus;

        cl_int err = clGetEventInfo(curDeviceStatus.WaitEvent,
                                    CL_EVENT_COMMAND_EXECUTION_STATUS,
                                    sizeof(eventStatus),
                                    &eventStatus,
                                    NULL);
        if(err != CL_SUCCESS)
        {
          throw clUtilException("ParallelFor_internal_error:_could_not_get_"
                                "event_info_" __WHERE__ "\n");
        }

        //If done, mark this device as available, notify the model, and
        //reset this device for more work
        if(eventStatus == CL_COMPLETE)
        {
          curDeviceStatus.Time2 = getTime();
          model.updateModel(curDeviceStatus);

          curDeviceStatus.IsBusy = false;

          clReleaseEvent(curDeviceStatus.WaitEvent);
          curDeviceStatus.WaitEvent = NULL;

          iterationsRemaining -= curDeviceStatus.Range.End -
                                 curDeviceStatus.Range.Start +
                                 1;

          //cout << "Iterations remaining " << iterationsRemaining << endl;
        }
      }
    }
  }

  Device::SetCurrentDevice(oldDeviceNum);
}
```

Appendix B

Source Code for PINA GetWork Function

```
IndexRange PINAScheduler::getWork(const size_t deviceGroup)
{
  IndexRange work;

  size_t& currentSampleRef = mCurrentSample[deviceGroup];
  size_t chunkSize = mChunkSize[deviceGroup];

  //If we haven't sampled everything with the current device, pull work from
  //The next sampling region
  while(currentSampleRef < mNumSamples)
  {
    //If this work region has no more work, move on
    if(mTasksRemaining[currentSampleRef].End <
       mTasksRemaining[currentSampleRef].Start)
    {
      currentSampleRef++;
      continue;
    }
    else
    {
      size_t iterationsRemaining = mTasksRemaining[currentSampleRef].End -
                                   mTasksRemaining[currentSampleRef].Start + 1;

      work.Start = mTasksRemaining[currentSampleRef].Start;
      work.End = chunkSize < iterationsRemaining ?
                 work.Start + chunkSize - 1:
                 mTasksRemaining[currentSampleRef].End;

      mTasksRemaining[currentSampleRef].Start = work.End + 1;

      currentSampleRef++;

      mIterationsRemaining -= work.End - work.Start + 1;

      return work;
    }
  }

  //Find the region of work where this device's norm speedup is greatest
  size_t bestSample = 0;
  double bestSpeedup = -1.0;

  for(size_t curSample = 0; curSample < mNumSamples; curSample++)
  {
    //Ignore this sample if there's no work left
    if(mTasksRemaining[curSample].Start > mTasksRemaining[curSample].End)
    {
      continue;
    }

    const Sample& thisSample = mModel[deviceGroup][curSample];
    double normSpeedup = 0.0;

    for(size_t curGroupID = 0;
```

```
        curGroupID < DeviceGroupInfo::Get().numGroups();
        curGroupID++)
    {
      //Don't look at speedup over ourself
      if(curGroupID == deviceGroup) { continue; }

      double thatTime;

      if(curSample < mNumSamples - 1)
      {
        thatTime = interpolate(mModel[curGroupID][curSample],
                               mModel[curGroupID][curSample + 1],
                               thisSample.Index);
      }
      else //Last sample region uses the mean as the right sample
      {
        Sample meanSample;
        double meanTime = mMeanIterationTime[curGroupID].TotalTime /
                          mMeanIterationTime[curGroupID].IterationsCompleted;

        meanSample.Index = mTasksRemaining[curSample].End;
        meanSample.Time = meanTime;

        thatTime = interpolate(mModel[curGroupID][curSample],
                               meanSample,
                               thisSample.Index);
      }

      double speedup = thatTime / thisSample.Time;

      normSpeedup += speedup * speedup;
    }

    normSpeedup = sqrt(normSpeedup);

    if(normSpeedup > bestSpeedup)
    {
      bestSpeedup = normSpeedup;
      bestSample = curSample;
    }
  }

  //Get work from the best sample
  work.Start = mTasksRemaining[bestSample].Start;
  work.End = work.Start + chunkSize <= mTasksRemaining[bestSample].End ?
    work.Start + chunkSize :
    mTasksRemaining[bestSample].End;

  mTasksRemaining[bestSample].Start = work.End + 1;

  mIterationsRemaining -= work.End - work.Start + 1;

  return work;
}
```

Vita

Rick Weber has always had a fascination with computers, writing a Missile Command clone for the TI-86 calculator during band class in high school. He received his BS, MS, and PhD in Computer Engineering at the University of Tennessee, focusing on high performance computing. His research area specifically focuses on using GPUs in scientific applications and heterogeneous computing. After graduating, Rick started working for Microsoft in their Advanced Information Processing division.